Titre: Title:	Lost in The Latent Space: Understanding, Auditing, and Evaluating Machine Learning Models for Code
Auteur: Author:	Vahid Majdinasab
Date:	2025
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Majdinasab, V. (2025). Lost in The Latent Space: Understanding, Auditing, and Evaluating Machine Learning Models for Code [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/70099/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/70099/
Directeurs de recherche: Advisors:	Foutse Khomh
Programme: Program:	génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Lost in The Latent Space: Understanding, Auditing, and Evaluating Machine Learning Models for Code

VAHID MAJDINASAB
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Novembre 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Lost in The Latent Space: Understanding, Auditing, and Evaluating Machine Learning Models for Code

présentée par **Vahid MAJDINASAB**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Christopher J. PAL, président
Foutse KHOMH, membre et directeur de recherche
Amir-massoud FARAHMAND, membre
David LO, membre externe

DEDICATION

For Aghajoon and Bibi...

ACKNOWLEDGEMENTS

I treated this part as an afterthought, something to be written only after everything else was done. Only when I finally began writing it did I find out that I had been living out one of literature's oldest tropes: a protagonist embarks on a quest in search of some kind of treasure. There is a map pointing the way, and it's often marked with *HIC SVNT DRACONES*. The road is fraught with trials meant to test the protagonist's worth, and on their way, they meet new friends who join them on their journey. There are always mentors; our protagonist and friends can never reach the treasure without someone to guide them after all. Arc after arc, the story goes on until the protagonist finally reaches the treasure, and ironically, for stories that are all about treasure, finding it is often the least interesting part. Most of the time, the treasure turns out to be less valuable than what the protagonist thought, but they are no longer the same person who started the quest either. The trials they went through, the guidance of their mentors, and the help of friends who accompanied them throughout the journey have changed them. It is fitting, then, that this trope is called "The real treasure was the friends we made along the way".

To my supervisor, Pr. Foutse Khomh. Thank you. None of the works in this thesis would have been possible without your guidance, patience, and push for excellence. You taught me how to think critically, question relentlessly, and aim high.

To my co-authors and collaborators. Thank you for helping shape this research and making it stronger through your contributions.

To my colleagues and fellow researchers in our group. Thank you. For these years that our lab became my second home, you became my second family.

To the *Alghonbelah* gang and *Sensei*, my oldest and dearest friends now scattered across the world. Though we are far apart, you have been with me through it all. Thank you.

To Dr. Amin Nikanjam, my mentor and friend. I don't know how to express how grateful I am for all your support, wisdom, kindness, and advice. No matter how I try to phrase it, it feels too small for what it really is. Thank you.

Finally, to my parents, always in my corner. You always cheered for me, celebrated each milestone with me, and walked beside me through every step. Words fail me here, too. For everything, thank you and thank you and thank you. Again and again and again.

A PhD may be a solitary quest, but it is an impossible journey to complete alone, and thanks to all of you, I can finally mark my map with *HIC ERANT DRACONES*.

RÉSUMÉ

L'intégration croissante de modèles d'apprentissage automatique dans l'ingénierie logicielle transforme de façon significative les processus d'ingénierie logicielle. Alors que les premières applications de tels modèles se limitaient à des outils fondés sur des règles, ciblant des tâches spécifiques et interprétables, l'essor de l'apprentissage profond et des grands modèles de langage a conduit à des systèmes puissants mais opaques, désormais employés pour automatiser des étapes centrales du processus d'ingénierie logicielle.

Ce passage d'un flux de travail compréhensible par l'humain et fondé sur des règles à des modèles de type boîte noire pilotés par les données a introduit des défis en matière de transparence, de fiabilité et d'auditabilité du processus d'ingénierie logicielle. Les pratiques traditionnelles telles que les tests unitaires, la relecture de code et l'analyse statique demeurent indispensables, mais elles ont été optimisées pour un développement déterministe, dirigé par des humains, avec des spécifications explicites et des justifications de conception traçables. À l'inverse, les modèles d'apprentissage automatique apprennent des représentations internes à partir des données et des objectifs, ce qui rend difficile l'interprétation, l'audit ou la garantie de leur comportement de la même manière. Il en résulte une lacune critique : à mesure que ces modèles sont de plus en plus déployés dans la chaîne d'ingénierie logicielle, l'absence de transparence et de garanties de confiance risque de miner la fiabilité, la sécurité et la confiance accordée au code généré.

Cette thèse s'attaque à ces défis dans le contexte des modèles d'apprentissage automatique entraînés sur du code. Nous proposons trois cadres distincts, chacun conçu pour traiter un aspect précis de la fiabilité et de la confiance : l'apprentissage de représentations, l'audit des modèles et l'évaluation des modèles.

Nous examinons d'abord ce que les modèles d'apprentissage automatique apprennent du code au cours de l'entraînement et la manière dont leurs représentations internes encodent des informations syntaxiques et sémantiques. Pour permettre une analyse systématique, nous introduisons *DeepCodeProbe*, un cadre d'inspection "probing" destiné à évaluer la structure et la qualité des représentations de code apprises. Ce travail apporte des éclairages sur la façon dont les représentations de code et la capacité des modèles influencent la rétention de caractéristiques de code essentielles, et il favorise une ingénierie des représentations et une conception de modèles plus interprétables et robustes.

Nous abordons ensuite le besoin d'un audit transparent des grands modèles de langage (GML) entraînés sur du code, d'autant que les jeux de données d'entraînement croissent en taille et

ne sont pas mis à disposition pour examen. Nous proposons *TraWiC*, un cadre permettant de détecter l'inclusion de jeux de données dans des GML entraînés sur du code, sans exiger l'accès ni aux jeux de données d'origine ni aux poids des modèles. Cette approche permet d'identifier d'éventuelles fuites de données, des violations de propriété intellectuelle et des phénomènes de mémorisation, contribuant ainsi à un développement et à un audit de modèles dignes de confiance.

Enfin, nous répondons au besoin de cadres de référence pour l'étalonnage "benchmarking" fiables, extensibles et adaptatifs pour les GML de code. Les benchmarks statiques ne parviennent pas à saisir de façon précise les faiblesses ni l'évolution des capacités des GML, car ils se concentrent uniquement sur des signaux binaires de succès/échec sur des tâches sous-jacentes, et ils sont susceptibles d'être incorporés dans les jeux de données d'entraînement des GML, ce qui gonfle artificiellement les résultats. Les benchmarks dynamiques existants s'appuient souvent sur d'autres GML pour juger les performances du modèle évalué ("LLM-as-judge"), ce qui introduit des biais, ou manquent d'une structure fondée sur des principes pour l'évaluation des tâches, ce qui engendre une forte variance entre exécutions. En outre, ils cherchent à tester plusieurs capacités à la fois (par exemple, la résolution d'un ticket), ce qui masque *comment* le modèle tente de résoudre la tâche et dissimule ses faiblesses en regroupant plusieurs exigences de raisonnement. Nous proposons *PrismBench*, une méthodologie d'évaluation dynamique nouvelle pour un benchmarking complet et adaptatif des capacités de génération de code des GML, afin de pallier les limites des approches statiques et dynamiques actuelles. *PrismBench* permet des diagnostics plus fiables et plus fins et soutient le développement de modèles robustes et de haute assurance pour la génération de code en ingénierie logicielle.

Ensemble, ces cadres constituent une chaîne permettant de sonder, auditer et soumettre à des tests de résistance systématiques les modèles d'apprentissage automatique destinés à l'ingénierie logicielle. Notre travail fournit à la fois des fondements théoriques et des outils pratiques pour comprendre ce que ces modèles apprennent, la manière dont leurs données d'entraînement peuvent être auditées, et comment leurs capacités et limites en conditions réelles peuvent être évaluées de façon rigoureuse. Ce faisant, il fait progresser l'état de l'art des modèles d'apprentissage automatique dignes de confiance pour l'ingénierie logicielle et propose des méthodes concrètes pour accroître la transparence et la fiabilité à mesure que ces systèmes deviennent partie intégrante du développement logiciel.

ABSTRACT

The growing integration of Machine Learning models into Software Engineering is reshaping the development, testing, and maintenance of software systems. While early applications of such models were limited to rule-based tools targeting specific, interpretable tasks, the advent of Deep Learning and Large Language Models has led to powerful but black-box systems that are now being employed to automate core stages of the Software Engineering process.

However, this shift from human-understandable, rule-driven workflows to black-box, data-driven models has introduced challenges regarding transparency, trustworthiness, and auditability of the Software Engineering workflow. In software development, code is expected to be explainable and accountable: the rationale and behavior of human-written code can be justified against explicit specifications, and quality-assurance practices such as unit testing, code review, and static analysis were developed to enable that accountability. Machine Learning models, by contrast, learn internal representations from data and objectives, making it difficult to interpret, audit, or guarantee their behavior in the same manner. This creates a critical gap: as these models are being increasingly deployed in the Software Engineering pipeline, a lack of transparency and trustworthiness risks undermining the reliability, security, and trust in the generated code.

This thesis aims to address the challenges described above in the context of Machine Learning models trained on code. We provide three separate frameworks, each designed to address a specific aspect of reliability and trustworthiness: model representation learning, model auditing, and model evaluation.

First, we investigate what Machine Learning models learn from code during training, and how their internal representations encode syntactic and semantic information. To enable systematic analysis, we introduce *DeepCodeProbe*, a probing framework for assessing the structure and quality of learned code representations. This work provides insights into how code representations and model capacity influence the retention of critical code features, supporting more interpretable and robust representation engineering and model design.

Next, we address the need for transparent auditing of LLMs trained on code, especially as training datasets grow in size and are not made available for inspection. We propose *TraWiC*, a framework for detecting dataset inclusion in LLMs trained on code without requiring access to original datasets or model weights. This enables the detection of potential data leakage, intellectual property violations, and memorization, thereby supporting trustworthy model development and audit.

Finally, we address the need for reliable, scalable, and adaptive benchmarking frameworks for code LLMs. Static benchmarks fail to capture LLMs' nuanced weaknesses or evolving capabilities by focusing only on binary success/failure signals on underlying tasks, and are prone to being incorporated in LLMs' training datasets, resulting in inflated benchmarking results. Existing dynamic benchmarks often rely on other LLMs to judge the performance of the model-under-test (i.e., LLM-as-judge), which introduces bias, or lack a principled structure for task evaluation, which introduces high variance between benchmarking runs. Furthermore, they attempt to test for multiple capabilities at the same time (e.g., resolving an issue), which masks *how* the model attempts to solve the task and hides model weaknesses by bundling multiple reasoning requirements into one. We propose *PrismBench*, a novel dynamic evaluation methodology for comprehensive and adaptive benchmarking of LLMs' code generation capabilities to address the shortcomings of current static and dynamic benchmarking approaches. This enables more reliable, fine-grained diagnostics and supports the development of robust, high-assurance models for code generation in Software Engineering.

Together, these frameworks form a pipeline for probing, auditing, and systematically stress-testing Machine Learning models for Software Engineering. Our work provides both theoretical foundations and practical tools for understanding what these models learn, how their training data can be audited, and how their real-world capabilities and limitations can be rigorously evaluated. In doing so, it advances the state of the art in trustworthy Machine Learning models for Software Engineering and offers concrete methods for increasing transparency and reliability as these systems become integral to software development.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

APR	Automated Program Repair
AST	Abstract Syntax Tree
CCD	Code Clone Detection
CFG	Control Flow Graph
CLPM	Code Pre-trained Language Models
CNN	Convolutional Neural Networks
DFG	Data Flow Graph
DL	Deep Learning
DNN	Deep Neural Network
FIM	Fill-In-the-Middle
GED	Graph Edit Distance
GNN	Graph Neural Network
GQA	Grouped-Query Attention
LLM	Large Language Model
LSTM	Long Short-Term Memory
MCTS	Monte-Carlo Tree Search
MDP	Markov Decision Process
MIA	Membership Inference Attack
ML	Machine Learning
MLM	Masked Language Model
NLM	Neural Language Models
NLP	Natural Language Processing
PDG	Program Dependence Tree
PEFT	Parameter Efficient Fine Tuning
POS	Part-Of-Speech
PT	Parse Tree
RL	Reinforcement Learning
RNN	Recurrent Neural Networks
SBFL	Spectrum-Based Fault Localization
SE	Software Engineering
SL	Supervised Learning
ST	Statement Tree
SVM	Support Vector Machine

SWA	Sliding Windows Attention
TBCNN	Tree-Based Convolutional Neural Network
TD	Temporal Difference
UL	Unsupervised Learning

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

Machine Learning (ML) is being increasingly integrated into nearly every stage of the Software Engineering (SE) process. Early uses centered around rule-based tools: single-task, interpretable systems designed for objectives such as variable type prediction or Code Clone Detection (CCD) [9]. The rise of Deep Learning (DL) led to more capable, task-specific ML models that significantly outperformed earlier rule-based approaches [10]. However, this came at the cost of transparency as these models are black boxes that learn internal statistical representations directly from training data and objectives, without inherent interpretability [11–13]. Despite this, such models are now widely used for tasks such as identifying vulnerable functions [14, 15], recommending large-scale refactoring [16], summarizing code behavior [17,18], and repairing broken functionality based on previously observed faults [19, 20].

The advent of Large Language Models (LLMs) has further accelerated the integration of black-box ML models in the SE pipeline. Initially, LLM-based systems such as GitHub Copilot [21], were deployed as assistants, helping developers autocomplete multi-line functions, generate documentation, or write unit tests. However, LLMs' capabilities are rapidly advancing [22, 23]. As such, LLMs and LLM-based systems are now used to generate entire codebases [24,25], produce comprehensive test suites [26], and even act as autonomous agents by diagnosing and fixing issues without direct developer supervision [27]. Currently, both task-specific models and general-purpose LLMs are being used in the SE development and deployment pipeline, from platform-specific assistants like Amazon Q [28] to coding agents such as OpenAI's Codex [29]. These systems are embedded across code editors, version control systems, CI/CD pipelines, and infrastructure-as-code workflows [9]. Therefore, ML models in SE are no longer limited to niche automation tasks and have become a foundational aspect of how modern software is built, tested, and maintained.

This widespread adoption has resulted in new challenges for software trustworthiness and reliability. Software development requires robustness as there are many points of subtle and critical failure, ranging from functional bugs and security vulnerabilities to compliance violations and cascading system failures [30]. Consequently, the SE field has developed a wide array of workflows such as unit tests, type checkers, linters, formal methods, and code reviews [31] to mitigate these risks through decades of research and experience. While these methods continue to evolve, they are all based on the assumption that code is written and reasoned about by humans and therefore remains understandable, traceable, and auditable.

The integration of ML models into SE breaks this assumption as these models' behaviors are learned, not explicitly designed, which makes tracing, debugging, and explaining their outputs difficult [9]. Therefore, unlike compilers or rule-based systems, current ML models offer no guarantees about consistency, correctness, or coverage [32]. This creates a growing mismatch: on one hand, these models offer significant boosts to developer productivity [33]; on the other, they introduce behavior that is difficult to inspect and verify as the size of the underlying codebases grows [34].

Despite these challenges, the adoption of ML in SE continues to accelerate [9, 10, 33]. Foundation models trained on massive corpora are already being deployed in production settings, powering developer tools and automation pipelines at scale [28]. However, as the integration of these models deepens, so does the need for a better understanding of their behaviors and limitations to develop and deploy trustworthy systems. This emphasizes the need for comprehensive frameworks tailored to the ML model lifecycle from training to deployment for SE tasks. Specifically, we need structured methods that expose what ML models learn from their training data, what data they rely on from their training datasets, and how they are evaluated. Building transparency across these dimensions is key to making ML for SE more reliable and trustworthy.

1.1 Research Questions

The objective of this thesis is to address the following central question:

> How can we enhance the transparency and trustworthiness of ML models trained on code for SE tasks?

In order to address this problem, we focus on three components of the ML training and deployment pipeline for SE:

- **Model training:** determining what ML models trained on code *learn* from their training data and how they represent this information in their latent space.
- **Model auditing:** determining what ML models trained on code *retain* from their training data without access to the underlying dataset or model weights, in order to enable dataset auditing, support IP protection, and ensure trust in the model's development process.
- **Model evaluation:** designing a *systematic* benchmarking framework for evaluating ML models on code in a manner that *adapts* to model behavior, provides a comprehen-

sive view of model capabilities, and reveals failure modes on end-to-end SE tasks such as code generation.

In this manner, we can decompose our central question into the following Research Questions (RQs):

RQa: *How can we inspect what ML models trained on code learn from their training data?*

To address this question, Chapter 4 presents the *DeepCodeProbe* framework for investigating the internal representations learned by ML models trained on code. This question aims to analyze what aspects of code structure and behavior are learned by small and mid-sized ML models during training.

RQb: *How can we audit LLMs trained on code to ensure data provenance without access to model internals?*

To address this question, Chapter 5 presents the *TraWiC* framework for dataset inclusion detection on LLMs trained on code. This question aims to provide a method for auditing whether a given codebase (or parts of it) was used in training, without requiring access to the original dataset or model weights.

RQc: *How can we evaluate LLMs on end-to-end code generation tasks and determine their capability ceilings?*

To address this question, Chapter 6 introduces the *PrismBench* framework, which dynamically evaluates LLMs on coding challenges by adapting to model performance in real time. This approach enables systematic exploration of model capabilities, edge cases, behavioral regressions, and performance inconsistencies and allows for a more reliable and comprehensive assessment of model capabilities beyond simple pass/fail metrics.

1.2 Contributions

This thesis contributes to the field of ML for SE by introducing frameworks aimed at improving the trustworthiness and reliability of ML models trained on code. Each contribution answers one of the core RQs defined above and addresses a specific gap in understanding, auditing, and evaluating such models. The key contributions of this thesis, our proposed frameworks, and subsequent analyses are as follows:

Contribution 1: *DeepCodeProbe*, a framework for probing internal representations learned by ML models trained on code.

While ML models achieve strong results on SE-related tasks, it remains unclear what syntactic or semantic features they learn from their training data. This lack of interpretability limits our ability to debug, improve, and deploy reliable and trustworthy ML models for SE tasks. This contribution provides insights into how different code representations used for training ML models, result in preserving essential code structure and syntax throughout the training process. This work answers three RQs:

RQa$_1$: *How can we assess the quality of different code representations in preserving essential syntactic properties for ML-based software engineering tasks?*

To answer this question, we introduce *DeepCodeProbe*, a probing framework for evaluating whether models trained on structured representations derived from code retain syntactic information. To ensure its reliability, we validate both the data representations and the extracted embeddings from the models under study, confirming that syntactic structure is preserved and encoded in the models' latent space.

RQa$_2$: *How do different representation choices affect the quality of learned features and task-specific requirements in ML models?*

To answer this question, we use *DeepCodeProbe* to compare how various code representations influence what models learn. Our analysis shows that structured representations help models focus on task-relevant features, while less expressive formats lead to weaker, less reliable abstractions.

RQa$_3$: *How does model capacity affect the quality of representation and the underlying downstream task?*

To answer this question, we probe models of varying capacities trained on the same representations to study how capacity correlates with representation quality. We find that while scaling improves syntax retention to a point, well-chosen representations are often more impactful than increased size, especially for maintaining generalization.

Contribution 2: *TraWiC*, a framework for auditing code LLMs for dataset inclusion detection and ensuring data provenance.

LLMs are trained on extremely large datasets scraped from the web, often with little to no transparency into what specific data was used during training. This raises concerns about

data leakage, intellectual property violations, and unintentional memorization of sensitive or copyrighted content. This contribution introduces a framework to audit code LLMs for dataset inclusion without requiring access to the original training data or model weights. This work answers three RQs:

RQb$_1$: *How can we detect dataset inclusion in LLMs without access to the training dataset or model weights?*

To answer this question, we introduce *TraWiC*, a framework for dataset inclusion detection on LLMs trained on code. We evaluate *TraWiC*'s performance on the dataset inclusion detection task, where the goal is to identify whether a given codebase was likely part of a model's training data. We compare its effectiveness against traditional code clone detection baselines used for this purpose and explore how different classifier choices impact inclusion detection results.

RQb$_2$: *How robust is TraWiC against data obfuscation techniques?*

To answer this question, we conduct a sensitivity analysis of *TraWiC*'s performance by testing it against various code obfuscation techniques and evaluating the importance of each feature type in the final detection outcome. This allows for determining the robustness of our framework under real-world conditions where data may be obfuscated or incomplete.

RQb$_3$: *How does TraWiC fail, and what can we learn from its failures?*

To answer this question, we investigate *TraWiC*'s failures (i.e., misclassifications) to understand when and why it produces false positives or false negatives. This includes examining both the model's behavior and the underlying properties of the code samples, providing insight into edge cases and the limits of our proposed framework.

Contribution 3: *PrismBench*, a dynamic benchmarking framework for LLMs on coding tasks.

Traditional code benchmarks rely on fixed datasets and static pass/fail metrics such as accuracy or pass@k. However, as LLMs grow in capability, these benchmarks increasingly fail to capture nuanced weaknesses or failure modes, which results in incorrect assessments of the LLM's capabilities. Therefore, this contribution aims to provide a dynamic benchmarking framework that adapts to model performance in real time. Rather than relying on static datasets or using LLMs for judging the model-under-test's outputs, *PrismBench* evaluates

models based on their actual task performance across evolving challenge scenarios, grounded in deterministic, reproducible measurements. This work answers three RQs:

RQc$_1$: *How can we design evaluation tasks that adapt to model capabilities and surface hidden failures?*

To answer this question, we introduce *PrismBench*, a dynamic, multi-agent benchmarking framework that models the evaluation process as a search problem over the space of all possible evaluation scenarios. Furthermore, instead of selecting evaluation scenarios blindly or from a static pool of pre-defined challenges, *PrismBench* dynamically generates tasks in response to the model-under-test's performance, which allows for probing edge cases, subtle weaknesses, and regressions that are otherwise missed in static benchmarks.

RQc$_2$: *How can we evaluate different models in a dynamically generated benchmark?*

To answer this question, we ground *PrismBench* in Reinforcement Learning (RL) by decoupling the benchmarking environment from the benchmarking process. We formalize a single, shared environment for all models and evaluate them by their traversal trajectories in this environment. To further reduce the risk of bias arising from LLM stochasticity, we design a multi-agent interaction sandbox that tests the models' capability in test generation, solution generation, and program repair, separately. Furthermore, we propose a set of performance metrics that can be used to compare models based on their performance throughout the search process. We analyze how models perform under identical dynamic conditions, compare their behavior across multiple failure modes, and show how *PrismBench* enables more fine-grained model capability comparison than static benchmarks.

RQc$_3$: *How can dynamic benchmarks provide insights into model performance, behavior, and failures?*

To answer this question, we examine the outputs collected during the evaluation process for each model and propose a set of diagnostic metrics that offer a comprehensive view of model performance and help identify patterns in how and why models fail.

Together, these three contributions form a pipeline for inspecting, auditing, and stress-testing ML models for SE tasks. The RQs developed in this thesis led to the publication/submission of the following research papers:

- **Majdinasab, V.**, Nikanjam, A., & Khomh, F. (2025). DeepCodeProbe: Evaluating Code Representation Quality in Models Trained on Code. Empirical Software Engineering, 30(6), 1-53. [35]
- **Majdinasab, V.**, Nikanjam, A., & Khomh, F. (2025). Trained without my consent: Detecting code inclusion in language models trained on code. ACM Transactions on Software Engineering and Methodology, 34(4), 1-46. [36]
- **Majdinasab, V.**, Nikanjam, A., & Khomh, F. (2025). PrismBench: Dynamic and Flexible Benchmarking of LLMs Code Generation with Monte Carlo Tree Search. Submitted to Advances in Neural Information Processing Systems (TMLR). [37]

The complete list of papers produced during the Ph.D study can be found in Annex 7.2.

1.3 Organization of the Thesis

This thesis is organized into seven chapters. Chapter 2 introduces the terminology and background knowledge necessary to understand the works presented in this thesis, namely, an essential background on ML for SE, code representations, probing, Membership Inference Attacks (MIA), and LLM benchmarking methodologies. Chapter 3 provides an overview of the literature and works related to the frameworks presented in this thesis. Chapter 4 presents our probing framework, *DeepCodeProbe*. Chapter 5 presents our dataset inclusion detection framework, *TraWiC*, followed by Chapter 6, which presents our dynamic, multi-agent framework for benchmarking LLMs, *PrismBench*. Finally, in Chapter 7, we conclude this thesis by summarizing our contributions and highlighting the limitations of the proposed frameworks before describing the possible avenues of research in reliable and trustworthy ML systems for SE.

CHAPTER 2 BACKGROUND KNOWLEDGE

This Chapter presents the general knowledge necessary to understand the contributions and the rest of the thesis. First, we will cover core concepts in ML, followed by a review of DL, with a focus on the principal architectures used for the models in this thesis. Given this thesis's focus on ML models trained on code, we cover how ML is applied in SE tasks alongside the different structured representations of code that are used for training such ML models. Next, we will explain probing, an approach used in Natural Language Processing (NLP) in order to understand what ML models learn from their training data. Afterwards, we will cover MIAs, which form the basis of dataset inclusion detection. Finally, we examine Large Language Models and how they are evaluated alongside the limitations of existing benchmarks for LLMs.

2.1 Machine Learning

Machine Learning refers to the study and development of algorithms that improve their performance on a task by learning from data, rather than relying on hardcoded logic [38]. The central idea is that, through exposure to data or repeated interaction with an environment, an ML model can identify patterns, optimize behavior, and generalize its knowledge to unseen instances of data.

Formally, the goal of an ML algorithm is to learn a function $f : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ is the input space and $\mathcal{Y}$ is the output space. This function is learned from a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$, where each $x_i \in \mathcal{X}$ is an input, and each $y_i \in \mathcal{Y}$ is the corresponding label or outcome. A loss/error function $\mathcal{L}(y, \hat{f}(x))$ quantifies the difference between the predicted output $\hat{f}(x)$ and the true output y. The learning objective then becomes finding a function $\hat{f}$ from the space of all possible functions $\mathcal{H}$ (i.e., the hypothesis space), such that the average loss over the entire dataset $\mathcal{D}$ is minimized:

$$\hat{f} = \arg\min_{f \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(y_i, f(x_i)).$$

Depending on the nature of the task (e.g., classification, clustering, control, etc) and the underlying data (e.g., image, sound, text, time series, etc.), ML algorithms fall into the following three categories.

2.1.1 Supervised Learning

Supervised Learning (SL) algorithms focus on learning a function that maps inputs to outputs based on the labeled training data [39]. This process includes iterating over the dataset $\mathcal{D}$ and learning which features in the input space $\mathcal{X}$ represent which label in the output space $\mathcal{Y}$. The learned function $\hat{f}$ can then be used to make predictions on data that is similar to what the model has seen during training. SL algorithms are used in applications such as image classification, speech recognition, natural language processing, and object detection.

2.1.2 Unsupervised Learning

In contrast to SL, where the goal is to learn a function that maps inputs to outputs based on *labeled* data, Unsupervised Learning (UL) algorithms are trained to find patterns and structures in the input space $\mathcal{X}$ without the presence of predefined labels in the dataset $\mathcal{D}$ [40] meaning that the output space $\mathcal{Y}$ is *inferred* from the features in the input space. Such patterns and structures can then be used to group similar data points or find important features to reduce the dimensionality of the underlying data. UL algorithms are used in applications such as cluster analysis, anomaly detection, and dimensionality reduction.

2.1.3 Reinforcement Learning

Unlike SL and UL, which rely on static datasets, RL algorithms are based on interactions between an agent and an environment. Instead of learning from a fixed set of labeled or unlabeled examples, the agent constructs its training data by acting in the environment and receiving feedback in the form of rewards [41]. This interaction is typically modeled as a Markov Decision Process (MDP) [41], defined as:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$$

where $\mathcal{S}$ is the set of all environment states, and $\mathcal{A}$ is the set of possible actions in the environment. $\mathcal{P}(s' \mid s, a)$ then defines probability of transitioning from state s to state s' by taking action a, and $\mathcal{R}(s, a)$ defines the reward the agent will receive by taking action a in state s.

The agent's goal is to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maps each state to an action in a manner that maximizes the expected cumulative reward over time. Unlike SL, the reward signal is often sparse and delayed, meaning the outcome of an action a in a state s affects the subsequent available states and corresponding rewards. As such, learning a good policy π

requires balancing between exploring the environment by taking new actions at s to discover other trajectories, and exploiting the current learned knowledge by taking actions at s which have resulted in high cumulative rewards in previous interactions. RL is particularly effective for sequential decision-making problems where the correct output is not available in advance, such as game playing, robotic control, and autonomous navigation.

2.2 Deep Learning

Deep Learning is a subfield of ML that focuses on models composed of multiple layers of parameterized transformations, which are referred to as Deep Neural Networks (DNNs) [11]. These models are capable of learning hierarchical representations of data, with each layer capturing increasingly abstract features from the outputs of the previous layers. The success of DL in recent years has been driven by a combination of increased computational resources, large-scale datasets, and algorithmic advances in optimization and regularization [42]. As such, DL models have become central to state-of-the-art performance in a wide range of domains.

2.2.1 Deep Neural Networks

A DNN can be formally defined as a function $f_\theta : \mathcal{X} \to \mathcal{Y}$ parameterized by a set of weights θ, and comprised of a sequence of L layers where each layer applies a linear transformation followed by a non-linear activation function:

$$h^{(l)} = \sigma(W^{(l)}h^{(l-1)} + b^{(l)}), \quad \text{for } l = 1, \ldots, L, \text{with } h^{(0)} = x \in \mathcal{X}, \quad f_\theta(x) = h^{(L)}.$$

where $W^{(l)}$ and $b^{(l)}$ are the weight matrix and bias vector of layer l, $\sigma(\cdot)$ is a non-linear activation function (e.g., ReLU, sigmoid), and $h^{(l)}$ is the output of the l^{th} layer.

One of the defining characteristics of DNNs is their ability to learn complex representations from data (i.e., representation learning) during training instead of relying on manually engineered features [11]. This capability resulted in high-performing models capable of learning from raw, high-dimensional inputs such as pixels, audio waveforms, or text tokens. There exists a variety of DNN architectures, each designed to address specific properties of the input space $\mathcal{X}$, task objectives, or inductive biases. While the general form of a DNN is a parameterized function f_θ, the structure of this function, in how the intermediate representations $h^{(l)}$ are calculated alongside the optimization objective, can differ significantly between different architectures and tasks. For example, Convolutional Neural Networks (CNNs) process image data by using weight sharing within layers [43], and Recurrent Neural Networks (RNNs) and

their variants process sequential data by incorporating temporal dependencies [11].

As this thesis is focused on ML models trained on code, we will focus on RNNs and the Transformer architecture, which are heavily used in this area.

2.2.2 Long Short-Term Memory Architecture

Recurrent Neural Networks are a class of DNNs designed to process sequential data by maintaining a hidden state that is updated at each time step. Formally, given an input sequence $\{x_t\}_{t=1}^{T}$, an RNN computes a sequence of hidden states $\{h_t\}_{t=1}^{T}$ where each hidden state h_t serves as a summary of the sequence up to time step t, allowing the model to make predictions based on previous inputs. However, standard RNNs struggle to capture long-term dependencies due to the *vanishing and exploding gradients* problem. In *vanishing gradients*, the changes to the weights become so small that they have little to no effect in improving the model's performance, while in *exploding gradients*, the changes become so big that the model's parameters never converge. This makes standard RNNs ineffective for tasks where relevant information appears far earlier in the sequence than where it is needed [11].

As such, Long Short-Term Memory (LSTM) networks were proposed to address this issue by introducing a more structured memory mechanism. Instead of a single hidden state, LSTMs maintain both a hidden state h_t and a cell state c_t, which is explicitly designed to preserve information over long time steps. At each time step t, the LSTM updates h_t and c_t using a set of "gates":

- The *forget gate* f_t determines what information to discard from the cell state.
- The *input gate* i_t controls what new information to add to the cell state.
- The *output gate* o_t decides how much of the updated cell state is exposed as output.

This gating mechanism enables LSTMs to retain relevant information over longer time steps than standard RNNs. However, training LSTMs is computationally expensive due to the number of parameters and gating operations.

2.2.3 Transformer Architecture

While LSTMs mitigate the vanishing gradient problem present in standard RNNs to a high degree, they still process input sequentially (i.e., each token is processed after the previous token has been processed), which limits parallelization and makes it difficult to capture long-range dependencies in very long sequences. Transformers were introduced to address these

limitations by discarding sequential processing and using the attention mechanisms to model relationships between tokens [44] instead of relying on hidden states.

Specifically, given a token sequence $X = [x_1, x_2, \ldots, x_n]$, transformers compute the relevance of each token x_i to every other token in the sequence using pairwise attention weights. This allows the model to capture global dependencies in the input regardless of the distance between tokens. Unlike LSTMs, which compress past information into a single hidden state h_t at each time step, transformers operate on the entire sequence simultaneously, enabling better understanding of the long-range context and taking advantage of parallelism during training.

Transformers have proven to be highly effective across a wide range of domains, including NLP and computer vision. Their scalability and ability to capture long-range dependencies have given rise to LLMs, which are trained on massive text corpora and exhibit strong capabilities in language understanding, generation, and reasoning, which we further discuss in Section 2.6.

2.3 Machine Learning for Software Engineering

Software engineering covers a wide range of activities throughout the software development life cycle, including design, implementation, testing, evolution, and maintenance [45]. Given the increasing sophistication and size of contemporary software systems [46], ensuring their correctness, efficiency, and long-term sustainability becomes increasingly challenging [47, 48]. These challenges span multiple dimensions, ranging from identifying bugs and optimizing performance to understanding legacy code and adapting systems to evolving requirements. As such, numerous ML-based approaches have been proposed to support and enhance key aspects of software engineering to address these challenges.

By training ML models on diverse software artifacts such as source code [49, 50], version control history [51], documentation [52], and software metrics [53], these models can assist developers in automating or improving performance across various software engineering tasks [10]. Specifically, ML-based approaches have been proposed for tasks such as:

Code generation: ML models have been proposed to generate code snippets from natural language descriptions, partial implementations, or other structured inputs. These models aim to automate routine programming tasks, accelerate development, and assist developers in rapidly prototyping solutions [54].

Defect prediction: ML models have been proposed to help developers anticipate potential

problems in specific areas of the codebase. This enables efficient resource allocation and proactive issue resolution [55–57].

Automated program repair: ML models have been proposed to help with automatically fixing common bugs. This can significantly reduce manual effort, minimize human errors, and save resources [20, 50, 58].

Code clone detection: ML models have been proposed to detect clones in the codebase by identifying duplicate or similar segments of code to facilitate refactoring efforts, ensuring consistency and reducing redundancy [59, 60].

Code summarization: Proposed ML models allow for understanding complex code logic, easier maintainability, and collaboration by generating abstract representations of code blocks [61, 62].

Comment generation: By producing descriptive comments, ML models allow for automatically increasing the intelligibility of code, ease of knowledge transfer, and support long-term maintenance activities [18, 63].

Code representation learning: Models trained to generate vector representations of code capture its syntactic and semantic properties, enabling transfer learning and downstream tasks such as code search or classification [64, 65].

Type inference: ML models can predict variable and function types in dynamically-typed languages, aiding code completion and static analysis [66, 67].

In the rest of this section, we will first cover structured representations of code that are used for training ML models for SE tasks. Afterwards, we present the ML-based approaches that are proposed for SE tasks that are selected to evaluate the approaches developed in this thesis.

2.3.1 Structured Representations of Code

Unlike natural language, source code is inherently structured, governed by strict syntactic and semantic rules. This structure can be explicitly captured and leveraged to improve ML models' understanding of program behavior. These structured representations of code aim to make these rules and relationships more explicit for ML models by encoding the underlying structure of programs as trees or graphs [68, 69].

Tree-Based Representations

Tree-based representations model the hierarchical structure of source code, capturing how program elements are nested within one another. Below, we go over the most common tree-based representations used for training ML models on SE tasks:

Parse Trees (PTs) preserve all tokens and grammar rules used to parse the source code, including elements like parentheses, commas, and formatting symbols. PTs are often large and extremely detailed, and are mainly used in settings where full syntactic fidelity is required, such as formatting tools or refactoring engines [70].

Abstract Syntax Trees (ASTs) model the syntactic structure of code by representing its tokens (e.g., variables, operators, keywords) as nodes in a tree. ASTs are created from PTs where each node corresponds to a syntactic construct (e.g., conditionals, loops, function definitions), and the edges represent their syntactic relationships. ASTs abstract away surface-level syntax details (e.g., formatting, punctuation, documentation, etc) from PTs while preserving code's core structure, making them widely used in tasks which require code modeling [71].

ASTs are widely used in code analysis tasks because they abstract away surface-level syntax while preserving program structure. In ML for SE, ASTs serve as input for models that need structural awareness, such as code summarization, translation, and clone detection. These representations can then be extracted by linearizing the AST, embedding node types, or using RNNs [10].

Graph-Based Representations

Graph-based representations extend tree structures by incorporating additional semantic and control-flow information. Below, we go over the most common graph-based representations used for training ML models on SE tasks:

Control Flow Graphs (CFGs) are used to model the flow of execution within a program. Nodes typically correspond to basic blocks—sequences of statements with a single entry and exit point. Edges between nodes then indicate possible control transfers (e.g., loops, conditionals, function returns). CFGs are used to model execution logic and are particularly useful in static analysis and security-focused tasks [10].

Data Flow Graphs (DFGs) model how data moves through a program by modeling dependencies between variables and operations. Nodes represent computations or variables, while edges represent data dependencies. DFGs are used in tasks like compiler optimization or symbolic execution [72].

Program Dependence Graphs (PDGs) combine both control and data dependencies in a single graph. PDGs are more expressive but also more complex; as such, they are mostly used for static analysis where a holistic view of program behavior is required [73].

Unlike ASTs, CFGs emphasize program behavior rather than syntax. This makes them useful for downstream tasks such as vulnerability detection, program analysis, and behavior modeling. In ML pipelines, CFGs are often encoded as graphs where node features and edge types are fed into Graph Neural Networks (GNNs) to capture control-flow sensitive patterns [10].

2.3.2 Code Clone Detection

Code Clone Detection (CCD) is an important activity in software maintenance as a fault in one project can also be present in other projects with similar code. Identifying and correcting these clones is essential to ensure overall software quality. Tasks such as software refactoring, quality analysis, and code optimization often necessitate the identification of semantically and syntactically similar code segments [74]. In CCD literature, clones are categorized into four types [75]:

- Type-1: Exact copies except for whitespace/comments.
- Type-2: Syntactically identical with different identifiers.
- Type-3: Similar fragments with added/removed statements.
- Type-4: Different syntax but similar semantics.

Detecting Type-1 and Type-2 clones is relatively straightforward, as they can be identified through syntactic comparisons. However, Type-3 and Type-4 clones pose a greater challenge, as they require capturing semantic similarities beyond surface-level syntax. Traditional approaches, such as text-based or token-based comparisons, often struggle with these more complex clone types [76]. To address this problem, the representation learning capabilities of DL models allow for the use of representations extracted from code such as ASTs, CFGs, and software metrics for CCD. By utilizing such representations, DL-based approaches such as FCCA [77], and Cclearner [78] have achieved high performance on CCD benchmarks.

2.3.3 Code Summarization and Comment Generation

Code summarization and comment generation are closely related tasks that aim to enhance code comprehension and maintainability. Code summarization involves generating concise natural language descriptions that capture the functional behavior or high-level semantics of a given code snippet [79]. On the other hand, comment generation focuses on producing comments that explain the purpose and functionality of specific code segments [80]. Allamanis et al. [17] train a CNN with an attention mechanism for code summarization. This approach leverages the strengths of CNNs in capturing local patterns while the attention mechanism helps in focusing on the most relevant parts of the code for generating accurate summaries. Another work done by Hu et al. [18] uses the ASTs constructed from code snippets to train a sequence-to-sequence model for comment generation for Java methods. By encoding the structural information from the AST, their model can better capture the code's semantics and produce more relevant comments tailored to the specific code segments. Wei et al. [81] trained two models for code generation and code summarization by initializing a sequence-to-sequence model for each task and training them alongside each other by using the loss of each network to improve the other. By effectively encoding the structural and semantic information from the code, these models can generate accurate and informative summaries and comments, facilitating code comprehension and maintainability.

2.3.4 Automated Program Repair

Automated Program Repair (APR) is aimed at reducing the manual effort required to fix bugs. Software defects not only compromise functionality but also pose significant challenges in terms of debugging time and maintenance costs [82]. APR techniques aim to automatically generate patches that fix defective code, thereby improving developer productivity and software reliability [45, 82]. Traditional APR techniques rely on predefined templates or search-based heuristics, which limit their generalization capability. In contrast, DL models can learn repair patterns directly from historical bug-fix data, enabling them to generalize to unseen code defects [20]. These models operate on various code representations (e.g., ASTs, token sequences, graph-based structures, etc.) to capture both syntactic and semantic aspects of buggy code [5, 6, 83].

Among ML-based APR approaches, several methods have been proposed to improve patch generation by leveraging code abstraction and token translation strategies. For example, DLFix [19] uses a novel fault-localization strategy to identify a buggy line and then extracts all associated AST nodes as a replacement subtree for patch generation. Next, it employs a novel abstraction strategy to rename variables in order to preserve variable types and avoid

variable name clashes by maintaining a mapping table to recover the actual variable names. Tufano et al. [20] introduced a code abstraction method for APR by using a lexer (based on ANTLR [84]) to tokenize the raw code. A parser is then used to determine the role of each identifier or literal and replaces each one with a unique ID to generate an abstracted source code version. Similarly, CoCoNut [58] applies an abstraction strategy that targets string and number literals. By abstracting these elements except for frequently occurring values (e.g., 0 and 1, which are used for boolean variables), CoCoNut simplifies code representations while retaining key semantic information, allowing for more effective learning of bug-fix patterns.

2.3.5 Code Representation Learning

Code representation learning approaches aim to transform source code into embeddings that capture both its syntactic structure and semantic meaning. Such embeddings can then be used for tasks such as code search, code summarization, and code classification [85].

For instance, CodeBERT [86] and StarCoder [87] use Masked Language Modeling (MLM) in order to predict masked tokens within tokenized code, which results in embeddings that can be used for downstream tasks. Other approaches incorporate additional structural information alongside tokenizing the raw code. such as GraphCodeBERT [88] and SynCoBERT [89], incorporate structural information by using data flow or ASTs to include explicit syntactical context into the embeddings. Additionally, contrastive learning methods [90] improve representation quality by either pairing different augmented views of the same code snippet (e.g., using dropout or token masking) in unimodal setups [65, 91], or by leveraging naturally aligned function-docstring pairs (similar to code translation and summarization) in bimodal setups [92].

2.3.6 Type Inference Prediction

Type inference is particularly useful for dynamically typed languages such as Python or JavaScript, where explicit type annotations are either missing or only partially documented. Automatically predicting the types of variables, function arguments, or return values can improve static analysis, code completion, and overall software reliability. Traditional rule-based approaches often struggle to handle context-sensitive or user-defined types, which limits their effectiveness. As such, multiple ML-based approaches have been proposed for this task. Raychev et al. [93] proposed JSNice, which predicts JavaScript types by modeling variable dependencies and learning probabilistic correlations across large codebases. Xu et al. [94] propose a probabilistic framework for Python that incorporates contextual cues such as attribute access patterns and identifier naming conventions. On the other hand, by

formulating type inference as a translation task from untyped to typed code, DeepTyper [95] uses a recurrent neural network in order to capture long-range dependencies and accurately predict variable types.

2.4 Probing in Natural Language Processing

Probing is used in NLP as a methodology for assessing the linguistic capabilities learned by ML models [96]. In general, probing involves evaluating how well the model embeddings capture syntactic [97], semantic [98], or other types of linguistic properties inherent in language data [99,100]. Probes are classifiers trained to predict specific linguistic properties (features) from embeddings extracted from a model given an input. These properties can include a word's part-of-speech (POS), its syntactic parent in a dependency tree, or whether two words refer to the same entity (coreference). Since these features are not explicitly provided to the model during training, the ability of a probe to recover them from the embeddings suggests that such information is implicitly encoded. Therefore, high classification accuracy indicates that the model has sufficiently captured the target linguistic property during its training [101]. NLP probes can be categorized into several types, each focusing on different linguistic aspects [102, 103]:

Syntactic probes: These probes assess whether the model representations capture syntactic properties of language, such as part-of-speech tags, constituent or dependency parsing structures, and word order information.

Semantic probes: These probes aim to evaluate the model's ability to encode semantic information, such as semantic roles, and lexical relations.

Coreference probes: Coreference resolution is the task of identifying and linking mentions that refer to the same entity within a text [104]. Probing for coreference capabilities helps in understanding if the model can capture long-range dependencies and resolve ambiguities.

By systematically probing for these different linguistic properties, researchers can gain insights into the strengths and weaknesses of various model architectures and investigate the specific representations learned by the models, increasing interpretability and reliability [102].

2.5 Membership Inference Attacks

One of the long-standing problems in ML is the tendency of models, especially large and overparameterized ones, to *memorize* portions of their training data rather than learn to generalize [105]. This behavior can pose significant privacy risks, particularly when the training data contains sensitive or proprietary information.

Membership Inference Attacks (MIAs) [106] have been shown to be effective in extracting information from ML models regarding their underlying training data. In an MIA, an adversary is given access to query a model M and attempts to determine whether a specific data record DR was included in M's training set. An MIA is considered successful if the attacker can correctly identify whether DR was included in M's training dataset $\mathcal{D}$ or not. MIAs exploit the memorization problem, which often results in models behaving differently on data they have seen during training compared to unseen data. For example, a model may produce lower loss [106, 107], higher confidence predictions [108, 109], more consistent outputs for training data [110, 111], or completely return an instance from its training dataset [112, 113].

2.6 Large Language Models

One of the highly active fields of research in ML is NLP. Here, research is focused on designing approaches that are capable of understanding natural language and generating responses that are contextually relevant, accurate, and coherent.

2.6.1 Neural Language Models and Large Language Models

Neural Language Models (NLMs) are trained to predict the probability of the next word (i.e., token) in a sequence by considering the previous tokens [114]. Early models such as ELMo [115] used stacked bidirectional LSTMs to generate context-aware word embeddings. Later, models like BERT [116] introduced the use of the Transformer encoder architecture to learn bidirectional representations based on the entire input sequence. These models were trained on large unlabeled corpora to learn word representations based on context (i.e., other tokens in the sequence) without any labeled data.

This process is known as *pre-training* in which a model is trained on an unlabeled dataset to learn word representations, language structure, syntax, and semantics [117, 118]. Pre-trained models can then be *fine-tuned* for specific downstream tasks by continuing their training on a labeled dataset designed for the task at hand. As such, by incorporating the Transformer architecture and self-attention mechanism [44], models such as GPT-2 [119] were trained using

the pre-training/fine-tuning paradigm alongside new architectures. Afterwards, models such as GPT-3 [120] were trained by mainly using similar architectures but scaling the size of the model alongside its training data. As the number of these models' parameters is extremely large, they are called *Large Language Models*.

While pre-training allows LLMs to learn general-purpose representations from large-scale corpora, these models often require further *fine-tuning* to perform well on specific downstream tasks. This process involves continued training on a smaller, labeled dataset tailored to a particular task or domain [121]. By doing so, model behavior is refined to better meet the requirements of the target use case, such as question answering, sentiment analysis, machine translation, or code generation [122]. Beyond performance improvements, fine-tuning is important for aligning models with user expectations, mitigating harmful behaviors, and integrating domain-specific knowledge [123].

2.6.2 Large Language Models for Code

In addition to natural language, code is often included in the training datasets of LLMs, even when the models are not specifically trained for code-related tasks, as it has been shown that including code in the model's training dataset improves its reasoning capabilities [21, 124].

LLMs fine-tuned on large datasets of code have demonstrated strong performance on SE tasks. Early LLMs for SE tasks were specifically focused on code generation. Models such as Codex [21], SantaCoder [125], and CodeLlama [126] were trained on massive code datasets and have shown high performance in code generation. In addition to code generation, many LLMs were specifically fine-tuned for other SE tasks. For instance, models like CodeT5 [127] and PLBART [128] were proposed for program repair, code clone detection, and code translation.

Regardless of the downstream task, all these models require a large corpus of high-quality code for pre-training and, even more importantly, fine-tuning [126, 129, 130]. High-quality code is typically defined as code with documentation that explains its structure and functionality, and studies have shown that the quality of the underlying training data can massively impact the performance of the LLM in downstream tasks [131–133]. The primary sources for collecting such data are GitHub and StackOverflow. Most open-source datasets, such as TheStack [134], are large aggregations of data collected from these platforms.

2.7 Benchmarking Large Language Models

To evaluate LLMs' performance, a wide range of metrics are used depending on the downstream task at hand. We cover the most commonly used metrics below.

Token-level Metrics measure the model's ability to predict the next token in a sequence or generate exact answers based on structured input-output tasks.

- Perplexity: Measures the likelihood of a sequence of tokens under the model. It is typically used in language modeling tasks to assess the model's ability to predict the next token in a sequence [119, 120].
- Exact Match and F1: Widely used in question answering benchmarks such as SQuAD [135] and Natural Questions [136]. Exact Match checks whether the predicted answer exactly matches the reference, while F1 measures the token-level overlap between them.

N-gram Overlap Metrics are used to compare the generated text to one or more reference texts based on surface-level overlap of n-grams or sequences.

- BLEU: Originally proposed as a metric for measuring models' performance in machine translation [137], BLEU evaluates n-gram precision between the generated and reference texts. It is also used in tasks such as summarization [138] and code generation [21].
- ROUGE: Is commonly used as a metric for measuring performance in text summarization tasks [139]. It is used to measure n-gram and sequence overlap.
- METEOR: Is another metric commonly used for measuring a model's performance in machine translation and text summarization. It measures alignment between generated and reference texts using stemming, synonym matching, and n-gram overlap. It has been used in translation and summarization tasks [140].

Task-Specific and Preference-Based Metrics are specifically designed for specialized evaluation settings such as code generation, alignment with human preferences, and factual correctness.

- Pass@k: Used in code generation tasks such as HumanEval [21], pass@k measures the probability that at least one of k generated solutions passes all provided test cases.

- Human Preference Ratings: Used in alignment-focused evaluations such as OpenAI's helpfulness and honesty benchmarks [141], where human annotators compare model responses based on quality, relevance, or other criteria.
- Multiple-Choice Accuracy: Used in knowledge and reasoning benchmarks such as MMLU [142], where models are evaluated based on their ability to answer multiple-choice questions across a wide range of domains.
- Truthfulness QA: Evaluates factual accuracy by asking models to answer questions that probe for misconceptions or false beliefs [143].

However, as LLMs continue to grow in scale and general-purpose capability (i.e., they are trained to handle many tasks such as translation, code generation, question answering, etc), reliably evaluating their performance has become increasingly difficult. Unlike models trained for specific tasks with well-defined inputs and outputs, general-purpose LLMs are designed to handle a large set of tasks (from single-word completions to multi-paragraph instructions) and can generate responses of arbitrary length and structure [144, 145]. This unbounded input and output space makes it difficult to define consistent evaluation criteria. Furthermore, LLM outputs vary in length, are sensitive to context, and subjective, with a single prompt potentially resulting in multiple potential completions depending on phrasing, temperature, or sampling strategy [146].

Alongside the challenges mentioned above, benchmarking recent LLMs is becoming more challenging. Leading LLMs are beginning to saturate widely-used benchmarks, achieving near-perfect scores that leave little room for meaningful comparison between different LLMs [22, 23, 147, 148]. Furthermore, as LLMs are trained on massive corpora scraped from a wide range of sources, many benchmarks risk being included in the training data, which undermines the benchmark's validity and the reported performance metrics [149]. Generalization is another challenge: as even when models perform well by standard metrics, these scores do not necessarily reflect deeper qualities such as reasoning, factual accuracy, or robustness to prompt variation [150, 151]. Small changes in phrasing or formatting can significantly affect model outputs, making evaluation highly sensitive to the underlying context [152].

2.8 Chapter Summary

This chapter provided the foundational knowledge required to understand the contributions and methodologies that will be presented throughout this thesis. We covered core ML concepts, covering supervised, unsupervised, and reinforcement learning paradigms, followed by an examination of DL architectures, with particular focus on LSTMs and Transformers that

form the backbone of modern code-related ML models. Afterwards, we discussed the ML applications in SE, exploring how structured code representations such as ASTs and CFGs enable effective model training for tasks including code clone detection, summarization, automated program repair, and type inference prediction. Next, we examined probing methodologies from NLP that allow us to understand what linguistic properties ML models learn during training, and discussed MIAs that exploit model memorization to recreate instances from their training datasets. We further analyzed LLMs, their application to SE-related tasks, and the significant challenges in benchmarking these models, including benchmark saturation, data contamination, and the difficulty of evaluating general-purpose capabilities.

These concepts establish the theoretical and practical foundation necessary for understanding the novel approaches to model probing, auditing, and evaluation methodologies that will be presented in subsequent chapters.

CHAPTER 3 LITERATURE REVIEW

This Chapter provides an overview of the works related to the frameworks presented in this thesis. First, we will discuss the proposed approaches for probing ML models (see Chapter 2.4). We then discuss the approaches used for MIAs (see Chapter 2.5) and dataset inclusion detection. Afterwards, we discuss the most used approaches for benchmarking LLMs (see Chapter 2.7). Finally, we conclude with an overview of how each of the frameworks presented in this thesis contributes to tackling the limitations of the works discussed below.

3.1 Probing ML Models Trained on Code

Given the success of BERT [116], researchers have conducted numerous studies on how BERT and BERT-based models learn and represent data for downstream tasks [153]. Although these studies primarily focus on general linguistic features rather than syntax-specific to code, the methodologies developed have been extensively applied to probe larger transformer-based models on how they learn and how they represent data in their latent space. Liu et al. [154] investigated the linguistic capabilities of pretrained models like ELMo, OpenAI's transformer, and BERT through seventeen probing tasks. They found that while these models perform well on most tasks, they struggle with complex linguistic challenges. Hewitt and Manning [97] introduced a structural probe to assess whether complete syntax trees are embedded within the linear transformations of neural network word representations (i.e., embeddings). They showed that models like ELMo and BERT encode syntax trees within their vector geometries. Miaschi et al. [155] compared the linguistic knowledge in the BERT's embeddings and Word2Vec. They used probing tasks targeting various sentence-level features and found that BERT's aggregated sentence representations hold comparable linguistic knowledge to Word2Vec.

With the advent of LLMs and their emerging capabilities, there is increasing interest in understanding how these models process information and make decisions based on their inputs. In the context of software engineering, researchers have adapted probing techniques originally used to understand the linguistic features that models learn from their training data to examine what models trained on code learn from their respective datasets. This has led to the creation of novel probing tasks specifically for large models trained on code. Wan et al. [156] have probed pre-trained language models trained on code to investigate if they learn meaningful abstractions related to programming constructs, control structures, and data types. They conducted structural analyses from three distinct perspectives: attention

analysis, probing on the word embeddings, and syntax tree induction. Their findings indicate that attention strongly aligns with the syntax structure of code, and that pre-trained models can preserve this syntax structure in their intermediate representations. Similarly, Troshin et al. [157] evaluated the capability of pre-trained code models, such as CodeBERT [86] and CodeT5 [127], in understanding source code, through diagnostic probing tasks, moving beyond task-specific models for code processing such as code generation and summarization. They introduced a variety of probing tasks to assess models' understanding of both syntactic structures and semantic properties of code, including namespaces, data flow, and semantic equivalence.

Karmakar et al. [158], investigated the extent to which pre-trained code models encode various characteristics of source code by designing and employing four diagnostic probes focusing on surface-level, syntactic, structural, and semantic information of code. Their methodology involves probing BERT [116], CodeBERT [86], CodeBERTa [159], and GraphCodeBERT [88] with tasks aimed at understanding the extent to which these models capture code-specific properties. Their findings show that these models can indeed capture the syntactic and semantic layers of code with varying effectiveness. However, all the models studied struggle to accurately comprehend the structural complexity of code, particularly the interconnected paths within it. Ma et al. [160] conducted a systematic probing of pre-trained code models (CodeBERT [86], CodeT5 [127], GraphCodeBERT [88], and UnixCoder [91]) alongside LLMs (CodeLlama [126], StarCoder [87], and CodeT5+ [161]) on their ability to learn code syntax and semantics. They investigate this through four probing tasks: (1) syntax node pair prediction, (2) token-level syntax role tagging, (3) semantic relation prediction (e.g., control/data dependencies), and (4) semantic propagation to reconstruct code structures such as ASTs, CFGs, and DFGs from the embeddings extracted from each model. They show that while these models are capable of representing code syntax well, their capabilities in representing code semantics (control and data flow) vary between different architectures and sizes. On the other hand, Karmakar et al. [162] evaluate code transformers by defining 15 probing tasks to assess how well these models capture token-level, structural, and semantic features of code. In contrast to [160], where the probing tasks are carried out on syntactic structures, their approach is designed for probing the models using token-based representations. Their results show that code-specific models like CodeBERT [86] and GraphCodeBERT [88] consistently outperform baseline BERT [116] on capturing token-level and structural features, yet struggle with semantic tasks, such as detecting subtle keyword misuse in code. Their results indicate that current pre-training methods require further refinement to effectively encode deeper semantic properties of code.

3.2 Membership Inference Attacks

Shokri et al. [106] studied how an attacker can identify the presence of a record in a model's training dataset by analyzing the model's output probabilities. Based on Shokri et al.'s work, many approaches have been proposed for both attacks on a model for extracting information about its training dataset and defenses in order to prevent the model from recreating sensitive records [106, 163–165]. Carlini et al. [166] attempted to extract the training data of GPT-2 by initializing the model with a sequence start token and having the model generate tokens until it generates an end-of-sequence token. They generate a large number of sequences in this fashion and remove samples that contain low likelihood. Their approach is based on the idea that samples with high likelihood are generated either because of trivial memorization or repeated sub-strings in their dataset. By doing so, they were able to extract information about individuals' contact information, news pieces, tweets, etc. In another work, Carlini et al. [167] study the quantity of memorization in LLMs. They show that by increasing the number of parameters of a model, the amount of memorized data increases as well. Chang et al. [168] attempt to probe ChatGPT and GPT-4 by using "name cloze" membership inference queries. In their approach, they query ChatGPT/GPT-4 by giving it a context (which in their work are texts extracted from novels), mask the token which contains a *name*, and inspect the output's similarity to the original token. The underlying idea is that without the model seeing the input during its training, the name should be impossible to predict. "Counterfactual Memorization" [169] is another approach proposed for studying the memorization issue in LLMs. Here, the changes in a model's predictions are characterized by omitting a particular document during its training. Even though this approach shows promise in studying a model's degree of memorization, it is very computationally expensive.

3.3 Dataset Inclusion Detection

Dataset inclusion detection approaches are built upon the principles of MIAs. Here, the goal is determining whether a data record of interest was included in the model's training dataset. Choi et al. [170] proposes an approach for verifying whether a particular model was trained on a particular dataset. Their approach consists of investigating multiple checkpoints produced during a model's training procedure. They show that if a model was trained on a dataset as reported by the model's creators, then the model's weights would "overfit" on the training dataset in the next checkpoint compared to a dataset that was not included in the training data. Even though their approach shows promising results, it requires having access to the model's architecture, weights, hyperparameters, and training data which is not

always available particularly for closed source models.

Zhang et al. [171] proposed a framework for dataset inclusion detection on Code Pre-trained Language Models (CLPMs). In this work, the authors consider multiple levels of access to a CLPM, namely:

- White-box access: complete access to the model and a large fraction of its training dataset.
- Gray-box access: complete access to the model's architecture and a small fraction of its training dataset.
- Black-box access: no access to either the model or the training dataset.

For dataset inclusion detection given only black-box access to the model, the authors provide two methods, namely, unimodal and bimodal calibration. In unimodal calibration, the code is perturbed and the differences in the model's output representations for different versions (e.g., uppercase vs lowercase) of the code snippet are observed. A calibration model, based on these differences, is used to infer the membership status of the code. In bimodal calibration, the calibration model's outputs are compared to the CPLM's outputs for the same input. This comparison is focused on identifying disparities between the two outputs, which are indicative of whether a given code snippet was used in training the CPLM or not. This method relies on the unique interaction between the natural language and programming language aspects of CPLMs, to make inferences about membership.

Finally, Yang et al. [172] examined the risk of membership leakage in LLMs trained on code. In their approach, they assume that a fraction of the data used to train an LLM on code is available. They train a surrogate model on both said fraction and another dataset that was not included in the original model's training. The surrogate model is supposed to mimic the behavior of the original model. Afterward, they train an MIA classifier based on the outputs of the surrogate model as an input to detect dataset inclusion. They study the results of varying degrees of membership leakage risks depending on factors such as the attacker's knowledge of the victim model, the model's training epochs, and the fraction of the training data that is known to the attacker. In their approach, they consider that at least a part of the training data of a model is available and it involves resource-intensive training of a surrogate model.

3.4 Benchmarking Large Language Models

As explained in Chapter 2.7, recent studies have demonstrated how current benchmarking approaches fall short in evaluating current LLMs. Specifically, Xu et al. [173], Roberts et al. [174], and Jiang et al. [175] show how even extensive attempts to control the training dataset does not prevent data contamination, either because filtering out undesired data is difficult given the training dataset's size [176] or because of models' increasing generalization capabilities and lack of diversity in benchmarks' challenges [177].

Crowd-sourced benchmarks such as Chatbot Arena [150] and Open LLM Leaderboard [178] have proven to be much more effective in evaluating LLMs but suffer from 1) a limited coverage of real-world challenges [179], and 2) user biases in preference ranking of the responses [150]. As a result, benchmarks such as ARC-AGI [180], SWE-bench [181], HELM [182], and HLE [183] have been proposed, which include either very difficult problems or data that the models could not have been trained on given the time of their release [184, 185]. The majority of leading LLM providers are currently using these benchmarks. However, with the rapid increase in LLM capabilities, these benchmarks are quickly becoming obsolete, as models achieve increasingly higher scores, rendering the evaluation metrics ineffective in distinguishing their performance [147, 148, 186].

Dynamic benchmarking approaches have been proposed to address the limitations mentioned above. LLM-as-Judge frameworks such as JudgeLM [187], AlpacaEval [188], and Shepherd [189] use one or multiple LLMs (based on benchmark performance or fine-tuned for specific evaluation criteria) to assess and rank the responses of the model under evaluation. However, as shown by Thankur et al. [190] given the judge models' capabilities and biases, offloading performance assessment to an LLM introduces challenges in the reproducibility and reliability of the evaluation. On the other hand, approaches such as Salad-bench [191], DELI [192], and Agent-as-a-judge [193] have been proposed for leveraging LLMs to generate novel and out-of-distribution evaluation scenarios based on existing static benchmarks.

Building on these approaches, frameworks such as TreeEval [194], DyVal [195], DyVal2 [196], NPHardEval [197], Benchmark Self-Evolving [198], and DARG [199] extend dynamic evaluation to provide a more systematic assessment of a model's capabilities. Notably, DyVal models each evaluation scenario as a DAG-based composition of reasoning tasks, guided by a graph generation algorithm, defined constraints, and translation into natural language. DyVal2 replaces the DAG with LLM-based agents to transform existing benchmarks into new challenges to evaluate LLMs' performance and generalization on multiple domains. DARG and TreeEval represent the evaluation processes as graphs or trees, analyzing how the model

navigates these structures. However, the stochasticity of LLMs and the variability in their performance introduce challenges in ensuring consistency and reliability in these evaluation methodologies, as shown by Blackwell et al. [200].

3.5 Chapter Summary

In this chapter, we discussed the related works on probing ML models trained on code, dataset inclusion detection, and benchmarking LLMs. The following chapters each present our proposed approaches for addressing the gaps in current literature. Specifically:

- In Chapter 4, we present *DeepCodeProbe*, a framework for probing small to mid-sized non-transformer-based models trained on structured representations of code. *DeepCodeProbe* addresses the gap in current literature about what such models learn from their training data and allows practitioners to make informed decisions about data representation and model architecture for training ML models on code.
- Chapter 5 presents our approach, *TraWiC*, for auditing LLMs trained on closed-source corpora for detecting dataset inclusion detection. Unlike current approaches, *TraWiC* does not require access to model weights or the underlying training dataset and has low resource overhead compared to DL based approaches.
- In Chapter 6, we present our dynamic LLM benchmarking framework, *PrismBench*. By framing LLM code generation benchmarking as an RL task, *PrismBench* addresses the shortcomings of both static and dynamic benchmarks by dynamically generating challenges based on observed model capability (therefore, reducing the risk of memorization) and grounding evaluations on objective execution of generated code (thereby, mitigating the risk of bias from LLM judgments).

CHAPTER 4 DeepCodeProbe: Evaluating Code Representation Quality in Models Trained on Code

The objective of this chapter is to study how the quality of syntactic representations affects what small ML models trained on code learn, and to evaluate this quality through probing (see Chapter 2.4). Given the increasing reliance on ML-based tools for SE tasks, understanding how data quality, specifically, the structure and abstraction of code representations, influences what such models learn is essential in order to establish reliable and trustworthy ML-based workflows. As discussed in Chapter 3, current probing approaches focus on large code LMs that are trained on token sequences for accomplishing multiple SE tasks (e.g., code generation, code summarization, program repair, etc). However, smaller, task-specific ML models are widely adopted given their low resource overheads for training and deployment (compared to large models), and more studied architectures (e.g., LSTMs, GNNs, etc). As such, there exists a gap for probing small to mid-size ML models trained on structured representations of code (see Chapter 2.3.1). To address this gap, we introduce *DeepCodeProbe*, a probing framework tailored to small and medium-sized models trained on structured code representations like ASTs and CFGs. We use *DeepCodeProbe* to probe a set of ML models trained for five different SE tasks and show that while these models often perform well, they rarely capture full programming syntax. Instead, they rely on abstracted syntactic patterns encoded in high-quality representations. By probing multiple layers and scaling model capacities, we analyze the trade-offs between model complexity, data representation, and learning outcomes. We show that when data representations are well-designed, smaller models can perform reliably without learning the full syntax of the programming language. This chapter provides both empirical findings and practical guidelines for designing efficient, reliable code-focused ML systems.

4.1 Introduction

Modern software systems face increasing complexity, which poses significant challenges across a variety of SE tasks [46]. ML models have emerged as powerful tools to address these challenges, assisting developers with tasks such as code clone detection [76], fault localization [10], and defect prediction [51]. These models process source code using diverse representations, ranging from simple text tokens to complex structures like ASTs and CFGs [10, 51, 76]. The quality of these representations critically impacts what the models learn [12, 201], and poor representations can introduce spurious patterns or blind spots [202,203], impacting reliability,

interpretability, and trustworthiness in safety-critical applications [204–206].

While prior work on ML data quality has focused on label noise and dataset bias [207], the impact of data representation quality, especially for code, is less understood. This gap is especially important given the field's increasing focus on large transformer-based models (see Chapter 2.6.2), which primarily operate on token-level representations and inputs. While effective, such models are computationally expensive, prone to hallucination, and often ill-suited for environments where interpretability, robustness, and verifiability are critical [208, 209]. In contrast, as described in Chapter 2.3, smaller task-specific models based on RNNs, GNNs, LSTMs, and tree-based architectures remain widely used due to their lower computational cost, reduced privacy risks, and compatibility with verification workflows. These non-transformer architectures have been the subject of interpretability research for much longer, making their internal representations more studied and their behavior more analyzable [210, 211]. However, even in these settings, where a non-transformer-based model is trained on structured representations like ASTs or CFGs, it is unclear whether these models truly learn syntax-relevant abstractions or merely memorize compressed patterns. Without methods to probe what these models internalize, it remains difficult to make informed decisions about representation design, model capacity, or reliability, especially in SE tasks with strict correctness requirements.

This chapter proposes a novel probing framework, *DeepCodeProbe*, to address these challenges for ML models trained on syntactic representations of code. By adapting and extending probing techniques originally developed in NLP, we construct a probing framework that can assess whether models trained on structured representations actually retain syntactic properties in their internal embeddings. Unlike existing probing methods that assume transformer-based architectures or token-level input (see Chapter 3.1), *DeepCodeProbe* is designed to work with smaller, task-specific models trained on explicit syntax structures like ASTs or CFGs. By using *DeepCodeProbe* on eight ML models trained on syntactic representations of code for five different SE objectives, we demonstrate that: (1) Models can achieve strong task performance without fully capturing programming language syntax, challenging common assumptions about what models need to learn. (2) High-quality structured representations like ASTs and CFGs enable reliable performance by inherently encoding important code properties, reducing the burden on models to learn these properties independently. (3) Well-chosen representations can be more important than model complexity for achieving reliable performance, highlighting new perspectives on the trade-offs in ML system design for software engineering.

To structure the study in this chapter, we define three RQs:

RQa$_1$: *How can we assess the quality of different code representations in preserving essential syntactic properties for ML-based software engineering tasks?*

To answer this question, we introduce *DeepCodeProbe*, a probing framework for evaluating whether models trained on structured representations like ASTs and CFGs retain syntactic information. To ensure its reliability, we validate both the data representations and the extracted embeddings, confirming that syntactic structure is preserved and encoded in the models' latent space.

RQa$_2$: *How do different representation choices affect the quality of learned features and task-specific requirements in ML models?*

To answer this question, we use *DeepCodeProbe* to compare how various code representations influence what models learn. Our analysis shows that structured representations help models focus on task-relevant features, while less expressive formats lead to weaker, less reliable abstractions.

RQa$_3$: *How does model capacity affect the quality of representation and the underlying downstream task?*

To answer this question, we probe models of varying capacities trained on the same representations to study how capacity correlates with representation quality. We find that while scaling improves syntax retention to a point, well-chosen representations are often more impactful than increased size, especially for maintaining generalization.

4.2 Methodology

In this section, we describe the methodology we followed to answer our RQs. For RQa$_1$, we first describe AST-Probe, which *DeepCodeProbe* is based on. Next, we describe our probing approach, *DeepCodeProbe*, and describe our validation methodology.

4.2.1 AST-Probe

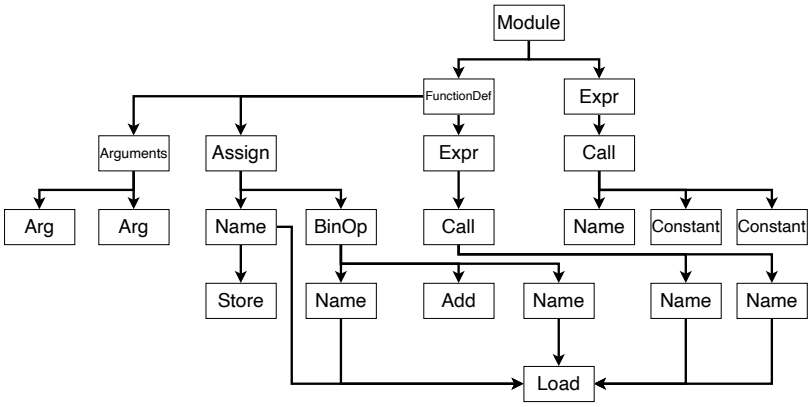
AST-Probe is designed to explore the latent space of LLMs and identify the subspace that encapsulates the syntactic information of programming languages [212]. AST-Probe consists of the following steps: the AST of an input code snippet is constructed and transformed into a binary tree. Next, the binary tree is represented as a tuple of $\langle d, c, u \rangle$ vectors. It must be noted that the transformation from AST to binary tree, and from binary tree to $\langle d, c, u \rangle$ is bi-directional, meaning that by having the $\langle d, c, u \rangle$ tuple, one can reconstruct the AST of the input code. Since the AST constructed from a code snippet is syntactically valid (i.e.,

AST can only be constructed if code is syntactically correct), predicting the $\langle d, c, u \rangle$ tuple from the representations extracted from the hidden layers of the model, given a code snippet as input, indicates that the model is capable of representing the syntax of the programming language in its latent space.

```
def sum(a, b):
    c = a + b
    print(c)

sum(1, 2)
```

(a) An example of a Python script



(b) The AST constructed for the script

Figure 4.1 An example of a Python script and its corresponding AST.

Figure 4.1b displays the AST extracted from a Python code snippet in Figure 4.1a. The $\langle d, c, u \rangle$ tuple is constructed by parsing the binary tree. In this tuple, d contains the structural information of the AST while c and u encode the labeling information. Algorithm 1 describes the construction of the $\langle d, c, u \rangle$tuple in detail.

After constructing the $\langle d, c, u \rangle$ of an input code snippet I, the embeddings of the model M for the code are extracted from different layers. Suppose that model M has n layers: $l_1, ..., l_n$. We denote the output of the l_{th} layer for input I as $M(I)_l$ with l being any layer between the second and the layer preceding the final layer (i.e., $2 < l < n-1$). After extracting $M(I)_l$ for an input code, the hidden representations are projected into a syntactic subspace S, and the probe is tasked with predicting the $\langle d, c, u \rangle$ from this projection. This process is repeated for all layers and all the code samples in the training data. If, for a given dataset of codes, the $\langle d, c, u \rangle$ tuples can be predicted by the probe with high accuracy, it indicates that the model is capable of representing the syntax of the programming language in its latent space.

4.2.2 DeepCodeProbe

As described in Chapter 2.4, probing is used in NLP to gain an understanding of what the model learns from its training data and analyze how the model makes a decision based on each input [102]. Furthermore, probing can also be used as an interpretability approach to

Algorithm 1 Constructing the $\langle d, c, u \rangle$ tuple for AST-Probe [212]

```
1: procedure TREE2TUPLE(node)
2:     if node is leaf then
3:         d ← []
4:         c ← []
5:         h ← 0
6:         if node has unary label then
7:             u ← [node.unary_label]
8:         else
9:             u ← θ
10:        end if
11:    else
12:        l, r ← children of node
13:        d_l, c_l, h_l ← TREE2TUPLE(l)
14:        d_r, c_r, h_r ← TREE2TUPLE(r)
15:        h ← max(h_l, h_r) + 1
16:        d ← d + [h] + d_l + d_r
17:        c ← c + [node.label] + c_l + c_r
18:        u ← u + u_l + u_r
19:    end if
20:    return d, c, u, h
21: end procedure
```

validate whether the model learns representations from the data required for solving the task at hand [96]. As we explain in detail in Chapter 3.1, previous works [156–158, 212] have used probing in order to explore the ability of LLMs to encode syntactic patterns of programming languages based on token-level representations within their latent spaces.

In this work, we are interested in DL models that are not as large as LLMs (i.e., language models trained on code that do not have hundreds of millions or billions of parameters) and focus on smaller models that are trained on explicit syntactic code structures (e.g., ASTs or CFGs) instead of raw code tokens. Our proposed approach is probing such models on their capability of representing the syntax of the programming languages in their latent space. However, most existing probing approaches are designed for large models and rely on high-dimensional embeddings derived from token-based transformer models. These methods assume properties such as architecture and input formats that do not generalize to smaller models trained on structured syntactic representations. Therefore, existing probing data representations tailored to token sequences cannot be directly applied to models trained on syntactic input. In order to address this gap, we base our approach, *DeepCodeProbe*, on AST-Probe [212], as described in Section 4.2.1, and introduce a novel data representation

tailored for probing models trained on structured syntactic inputs.

Feature Representation and Embedding Extraction

By basing our probing approach on the same principles of AST-Probe, we first extract a syntactically valid representation from the input code and construct a corresponding $\langle d, c, u \rangle$ tuple from it. However, unlike AST-Probe, in our approach, we do not derive the $\langle d, c, u \rangle$ tuple from the binary tree and instead build it directly from either the AST or the CFG extracted from the input code. We frame our probing approach as such since the binary tree of an AST or CFG would be very large and therefore, a $\langle d, c, u \rangle$ tuple constructed from it would be extremely high-dimensional. Since we are interested in models that are much smaller than LLMs and trained for specific downstream tasks, probing these models for such high-dimensional representations will result in sparse representations of vectors (e.g., deriving a representation with 4,000+ features from an input with only 128 features). This sparsity would make the probe extremely sensitive to noise [11]. Given that the probe's capacity should be kept as small as possible (i.e. the probe should have significantly fewer parameters than the model being probed) [97, 102, 212], such sparse representations of vectors will result in non-convergence because of the accumulation of small effects of noise in high dimensional data [213, 214]. Therefore, we define the $\langle d, c, u \rangle$ tuple construction from the AST or CFG of a given code as follows:

- d: The position of each node in the AST/CFG in sequential order level by level (breadth-wise) from left to right, as shown in Figure 4.2.
- c: Position of the children of each node in sequential order for ASTs and the connections between each node in CFG.
- u: Vector representation of each node in the AST/CFG extracted from the model's data processing approach.

Algorithm 2 describes an overview of our probing approach. Similar to AST-Probe, extracting the $\langle d, c, u \rangle$ tuple from an AST/CFG is bi-directional, meaning that the $\langle d, c, u \rangle$ tuple extracted from the AST/CFG of a code can be used to reconstruct the corresponding AST/CFG. As this conversion is bi-directional, we infer that if the probe can predict the $\langle d, c, u \rangle$ tuple from the embedding extracted from a model, then the model is capable of representing the syntax of the programming language in its latent space. In the same manner, we consider low performance on predicting the $\langle d, c, u \rangle$ tuple, as the model's incapability to represent the syntax of the programming language in its latent space [101, 212].

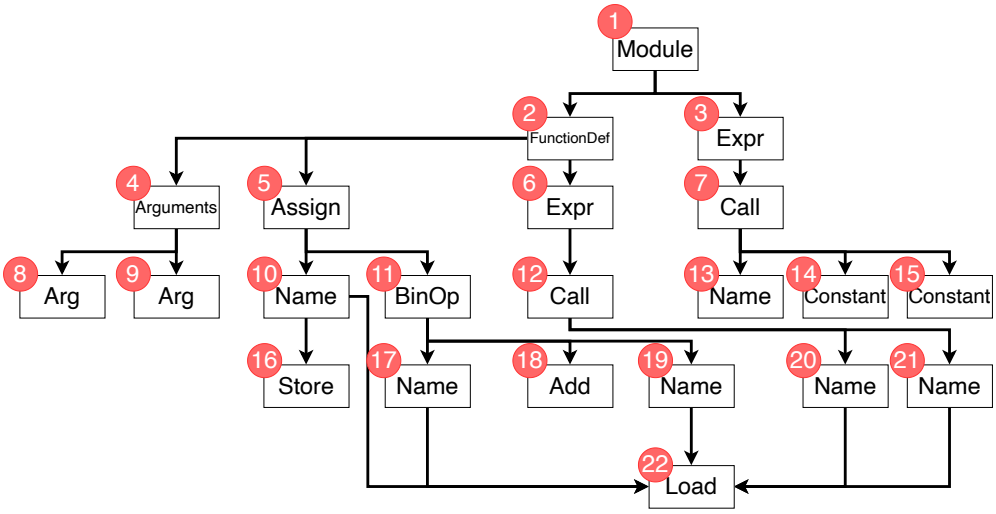


Figure 4.2 Annotated AST

Figure 4.2 displays the annotated AST that is used for $\langle d, c, u \rangle$ tuple construction (the code snippet is shown in Figure 4.1a and the constructed AST in Figure 4.1b). The red circles above each node indicate the assigned index with the indexing being done level by level (breadth-wise) from left to right. Following this running example, Listing 1 displays the constructed $\langle d, c, u \rangle$ tuple which will be used for probing the model.

Validating *DeepCodeProbe*

To ensure the reliability and accuracy of our probing task, we validate our proposed approach through the following three stages:

- *Validation of the data representation method:* We validate that the $\langle d, c, u \rangle$ tuple constructed for each AST/CFG contains information on the programming language's syntax.
- *Validation of the extracted embeddings:* We validate that the embeddings extracted from the model contain enough *distinctive* information from the input. This is crucial to ensure that embeddings extracted from the model, for codes with dissimilar ASTs/CFGs are distinctly different from those extracted for codes with similar ASTs/CFGs.
- *Validation of the embedded information:* We validate that the models under study learn a relevant representation from their training data for the task for which they have been trained. This is crucial to ensure that the models under study have not simply memorized their training data.

Algorithm 2 *DeepCodeProbe*

Require: Input code script SCRIPT
Ensure: Extracted DCU tuple DCU
procedure ExtractDCU(SCRIPT)
    if MODEL_INPUT_TYPE = AST **then**
        tree ← ConstructAST(SCRIPT)
    else if MODEL_INPUT_TYPE = CFG **then**
        tree ← ConstructCFG(SCRIPT)
    end if
    function Tree2Tuple(tree)
        $d, c, u \leftarrow$ Extract node positions, child indices, and node features from tree
        return (d, c, u)
    end function
    DCU ← Tree2Tuple(tree)
    return DCU
end procedure
Require: Trained model M, training data TRAINING_DATA
Ensure: Probing results PROBING_RESULTS
procedure ProbeModel(M, TRAINING_DATA)
    Probe ← InitializeProbe()
    PROBING_RESULTS ← Empty list
    for code ∈ TRAINING_DATA **do**
        DCU ← ExtractDCU(code)
        embeddings ← M(code)
        Predicted_DCU ← Probe(embeddings)
        $(\text{accuracy}_d, \text{accuracy}_c, \text{accuracy}_u) \leftarrow$ Compare(DCU, Predicted_DCU)
        Append $(\text{accuracy}_d, \text{accuracy}_c, \text{accuracy}_u)$ to PROBING_RESULTS
    end for
    return PROBING_RESULTS
end procedure

This three-step validation process is necessary because if the data representation approach or the extracted embeddings do not contain the information for which the model is being probed for, the results of the probing process will be invalid since we would be probing the models for information that is not available in their latent space. Furthermore, if a model's embeddings do not change to reflect the task for which it has been trained (before and after training), these embeddings cannot be used for probing, since they would not contain enough information. In the following, we explain each of the aforementioned validation stages in more detail.

Validating Data Representation

DeepCodeProbe is designed for probing models that use syntactically valid constructs derived

```
d = [ 1, 2, 3, 4, 5, 6, ....., 22 ]
c = [
        (2, 3) # children of node 1.
        (4, 5, 6) # children of node 2,
        (7) # children of node 3,
        (8, 9) # children of node 4,
        .......
    ]
u = [
        1 #Label of the ''Module'' node according to the model under study,
        54, #Label of the ''FunctionDef'' node according to the model under study,
        32, #Label of the ''Expr'' node according to the model under study,
        ......
    ]
```

Listing 1: $\langle d, c, u \rangle$ tuple example for AST probing.

from code to accomplish their task. As detailed in Section 4.2.2, we construct a $\langle d, c, u \rangle$ tuple for each code snippet with the d and c tuples being derived directly from the AST/CFG of the code, and the u tuple being extracted from the model under study's own internal encoding process. Previous works have shown that AST/CFG representations of code can be used to detect similarity between codes, as code clones of Types 1, 2, and weakly Type 3 have similar AST/CFG structures [215–218]. Given this, and considering how $\langle d, c, u \rangle$ tuples are constructed, a valid representation should preserve these similarity patterns. Specifically, the $\langle d, c, u \rangle$ tuples from clones should be similar to one another, while tuples from non-clones code should be dissimilar in comparison. Given that the d and c components directly encode syntactic information from the code's AST/CFG, these structural differences between code snippets should be explicitly captured as well. The u component, extracted from the model under study's data processing approach (as shown in Listing 1), should align with these patterns, as it is trained on AST/CFG inputs and each u label corresponds to elements in d and c.

Algorithm 3 details the process of constructing and validating $\langle d, c, u \rangle$ tuples. Specifically, we measure the cosine similarity between $\langle d, c, u \rangle$ tuples of clone and non-clone pairs. If our representation is valid, clone pairs should have significantly higher similarity scores in comparison to non-clone pairs. Importantly, as the d and c components are derived directly from the AST/CFG, they are invariant across models. That is, given a code snippet, its d and c values remain fixed regardless of the model under study. Only the u component, which captures model-specific embeddings, varies between models. This ensures that the structural integrity of the code is consistently encoded via d and c, while u reflects how different mod-

Algorithm 3 Creating and Validating $\langle d, c, u \rangle$ Tuples

Require: A set of code snippets $\mathcal{S}$ and a model under study M
Ensure: Similarity scores between $\langle d, c, u \rangle$tuples for clone and non-clone code pairs

procedure BUILDANDVALIDATEDCUTUPLES($\mathcal{S}, M$)
  Initialize empty dictionaries: DCU_Tuples, $SimilarityScores$
  function EXTRACTDC($code$)
    $ast/cfg \leftarrow$ Parse $code$ into AST/CFG
    return TREE2TUPLE(ast/cfg).{d, c} ▷ As defined in Algorithm 4 and 5 for M
  end function
  function EXTRACTU($code, M$)
    $modelInput \leftarrow$ Preprocess $code$ as required by M
    $ULabels \leftarrow$ Pass $modelInput$ through M's embedding layer
    return Labels for each node in AST/CFG
  end function
  for $code \in \mathcal{S}$ **do**
    $(d, c) \leftarrow$ EXTRACTDC($code$)
    $u \leftarrow$ EXTRACTU($code, M$)
    $DCU\_Tuples[code] \leftarrow \{d : d, c : c, u : u\}$
  end for
  function COSINESIMILARITY($vec1, vec2$)
    return $\frac{vec1 \cdot vec2}{\|vec1\| \|vec2\|}$
  end function
  for all $(codeA, codeB)$ in ClonePairs ∪ NonClonePairs **do**
    $(d_A, c_A, u_A) \leftarrow DCU\_Tuples[codeA]$
    $(d_B, c_B, u_B) \leftarrow DCU\_Tuples[codeB]$
    $similarity_d \leftarrow$ COSINESIMILARITY(d_A, d_B)
    $similarity_c \leftarrow$ COSINESIMILARITY(c_A, c_B)
    $similarity_u \leftarrow$ COSINESIMILARITY(u_A, u_B)
    $SimilarityScores[(codeA, codeB)] \leftarrow (similarity_d, similarity_c, similarity_u)$
  end for
  return $SimilarityScores$
end procedure

els interpret and represent these structures. Given that our $\langle d, c, u \rangle$ tuple conversions are bidirectional (i.e., the original AST/CFG can be reconstructed from them), this validation approach also ensures that our representation preserves both structural consistency through d and c tuples and model-learned semantics through the u tuples.

Validating The Extracted Embeddings

Previous works have shown that semantic similarity between data points is reflected in embedding proximity in DL models (i.e. if two inputs are similar, then their representations in the model's latent space are similar) [219, 220]. Therefore, in order to validate the extracted embeddings we follow a similar approach to validating the data representation. In

this step, instead of comparing the similarity of $\langle d, c, u \rangle$ tuples, we compare the similarity of the extracted embeddings. We use the same code clone pairs constructed for validating the data representation. As such, we input code clone and non-clone pairs to each model and extract the corresponding embeddings for each input. Afterward, we use cosine similarity to measure the distance of the model's embeddings between code clones and non-clones. Given that the models under study are trained on syntactically valid representations of code, then regardless of the task that they are trained for, we expect that the extracted embeddings for code tuples that are syntactically similar to be similar to each other compared to code tuples that are not. In other words, the cosine similarity for the embedding of similar codes should be significantly higher than the cosine similarity of dissimilar codes for the extracted embeddings to be valid.

Validating The Embedded Information

In probing literature, it is standard to show that the probe itself is incapable of learning representations from the extracted embeddings and instead only exposes information that already exists within the model's latent space [97, 156, 212]. This is done by training the probe on embeddings extracted from randomly initialized versions of the model (i.e. models before training) and embeddings extracted from the model after it has been trained. If there exists a significant difference in the probe's performance in predicting the property of interest, then it is assumed that the probe is incapable of learning representations from the embeddings itself [101–103, 212]. In our context, where we are probing small models trained on syntactic representations of code, we need to ensure that the probe's validity does not rely on the models' capacity to learn syntax. To address this, we propose an alternative validation method focusing on the similarity of embeddings for code clones, regardless of the models' syntax learning capacity. Given that the models are trained on syntactic representations of code, we extract embeddings for code clone tuples from both the trained and randomly initialized versions of the models. We then compare the cosine similarity scores for code clones from the trained model against those from the randomly initialized model. Higher similarity scores for the trained model would suggest that the model has learned some form of meaningful representations, even if not explicitly syntactic. Next, we train the probe on these embeddings and assess its performance. A significant difference in the probe's performance between embeddings from the trained and randomly initialized models would indicate that the probe is uncovering the existing structure in the embeddings rather than learning it during the probing process.

This comprehensive validation methodology ensures that our probe operates on reliable data

representations and embeddings. By validating the data representation, we confirm that our $\langle d, c, u \rangle$ tuples accurately capture the syntactic structures of codes. Through validating the extracted embeddings, we ensure that the models reflect meaningful semantic similarities. Finally, by validating the probe, we establish that it exposes the pre-existing information within the model's latent space without learning representations itself. These steps collectively validate the data representation and the probing approach, allowing us to analyze and draw conclusions about the models' capabilities in learning the syntactic properties of code.

4.3 Experimental Design

This section outlines the models included in our study and details the construction of the probing dataset for each model. To that end, we remind our RQs and their objectives:

RQa$_1$: *How can we assess the quality of different code representations in preserving essential syntactic properties for ML-based software engineering tasks?*

To answer this question, we introduce *DeepCodeProbe*, a probing framework for evaluating whether models trained on structured representations like ASTs and CFGs retain syntactic information. To ensure its reliability, we validate both the data representations and the extracted embeddings, confirming that syntactic structure is preserved and encoded in the models' latent space.

RQa$_2$: *How do different representation choices affect the quality of learned features and task-specific requirements in ML models?*

To answer this question, we use *DeepCodeProbe* to compare how various code representations influence what models learn. Our analysis shows that structured representations help models focus on task-relevant features, while less expressive formats lead to weaker, less reliable abstractions.

RQa$_3$: *How does model capacity affect the quality of representation and the underlying downstream task?*

To answer this question, we probe models of varying capacities trained on the same representations to study how capacity correlates with representation quality. We find that while scaling improves syntax retention to a point, well-chosen representations are often more impactful than increased size, especially for maintaining generalization.

We detail the models under study in Section 4.3.1 alongside the probing dataset construction for each model in Section 4.3.2. Next, we describe the design of the probes used in our analyses

for RQa_1 and RQa_2 in Section 4.3.3. Finally, we present the methodology for scaling each model's capacity to support our analysis in RQa_3 in Section 4.3.4.

4.3.1 Models Under Study

In this work, we are interested in models trained on code for software engineering tasks. As such, to answer our RQs, we select five tasks in software engineering for which learning the syntax of the programming language is considered to be important: code clone detection, code summarization, automated program repair, code representation learning, and type inference prediction. For each of these tasks, we select models based on the following criteria:

- They are not large models (i.e., they have below 100 million trainable parameters.)
- They are trained on syntactically valid representations of code, such as ASTs or CFGs, rather than raw code or code artifacts as text.
- They demonstrate high performance on benchmarks specifically designed for the tasks they have been trained for.

Considering the criteria defined above, we select the following models to assess our probing approach, *DeepCodeProbe*:

AST-NN

Zhang et. al. [1] proposed AST-NN as a novel approach for representing the AST extracted from code as inputs for DL models. In their approach, they break down the AST into Statement Trees (ST) to address the issues of token limitations and long-term dependencies in DL models, which arise from the large size of ASTs. This process involves a preorder traversal of the AST to identify and separate statement nodes defined by the programming language. For nested statements, statement headers and included statements are split to form individual statement trees. These trees, which may have multiple children (i.e., multi-way trees), represent the lexical and syntactical structure of individual statements, excluding sub-statements that belong to nested structures within the statement. This process reduces an AST into multiple smaller STs. Additionally, AST-NN employs a Word2Vec model [221], to encode the labeling information for each node in the AST by learning vector representations of words from their context in a corpus.

In AST-NN, the input code is broken down into STs to be used for two tasks, namely, source code classification and CCD. In this task, the STs extracted from two pieces of code are used

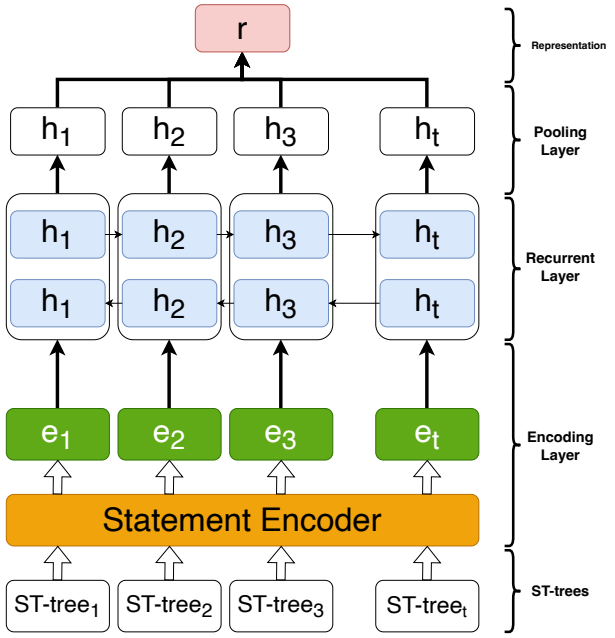


Figure 4.3 Model architecture for AST-NN [1]

as inputs for a DNN comprising an encoding layer, a recurrent layer, and a pooling layer. Figure 4.3 provides an overview of the AST-NN model architecture. The output of the representation layer is a binary value indicating whether two input codes are clones. The authors of AST-NN reported state-of-the-art clone detection results on the BigCloneBench [222] and OJClone [223] datasets for C and Java programming languages using their proposed approach.

FuncGNN

Similar to AST-NN, Nair et. al. [2] proposed FuncGNN, a graph neural network that predicts the Graph Edit Distance (GED) between the CFGs of two code snippets to identify clone pairs. FuncGNN, as presented in Figure 4.4, operates by extracting CFGs from input code snippets and utilizing a statement-level tokenization approach, which distinguishes it from the word-level tokenization used by other models under study, such as AST-NN, SummarizationTF, and CodeSumDRL. The core assumption of FuncGNN is that code clone pairs will exhibit a lower GED compared to non-clone pairs, leveraging this metric to determine whether two code samples are clones.

FuncGNN's methodology combines a top-down approach for embedding the global graph information with a bottom-up one for atomic-level node comparison to catch similarities between operations. The top-down approach generates an embedding that captures the overall structure and control flow of a CFG by incorporating an attention mechanism to find

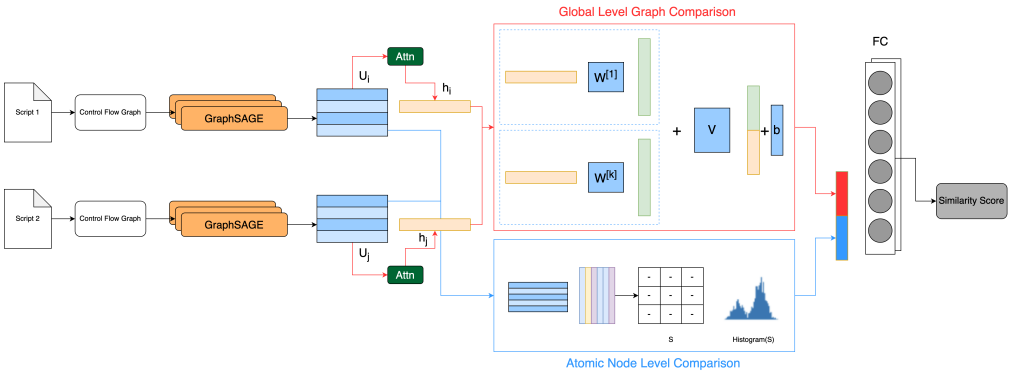


Figure 4.4 Model architecture for FuncGNN [2]

nodes in the CFG that have high semantic similarity. On the other hand, the bottom-up approach focuses on the comparison of node-level similarities within the CFGs to capture the similarities in program operations. By incorporating both approaches, FuncGNN predicts the GED between two input CFGs.

SummarizationTF

The approach proposed by Shido et al. [3] extends the Tree-LSTM model [224] to handle source code summarization more effectively by introducing the Multi-way Tree-LSTM. This extension allows the model to process ASTs that have nodes with an arbitrary number of ordered children, which is a common feature in source code. In this approach, the AST of an input code is extracted, and then each node in the AST is represented by a vector of fixed dimensions. These vectors are then used as input to the Multi-way Tree-LSTM, and an LSTM decoder is then used to generate natural language summaries. Figure 4.5 presents an overview of SummarizationTF's operation.

CodeSumDRL

Wan et. al. [4] proposed an approach for source code summarization based on deep reinforcement learning. Their proposed approach integrates the AST structure and sequential content of code snippets as inputs to an actor-critic architecture as presented in Figure 4.6. The actor-network suggests the next best word to summarize the code based on the current state, providing local guidance, while the critic-network assesses the possibility of occurrence of all possible next states. Initially, both networks are pre-trained using supervised learning (i.e., code and documentation tuples), using cross-entropy and mean square as loss functions, for the actor and the critic, respectively. Afterward, both networks are further trained

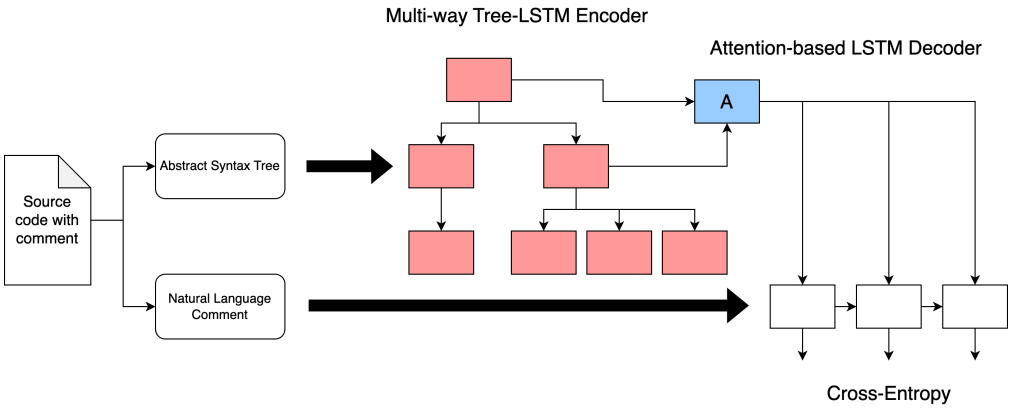


Figure 4.5 Model architecture for SummarizationTF [3]

through policy gradient methods using the BLEU metric as the advantage reward to improve the model's performance.

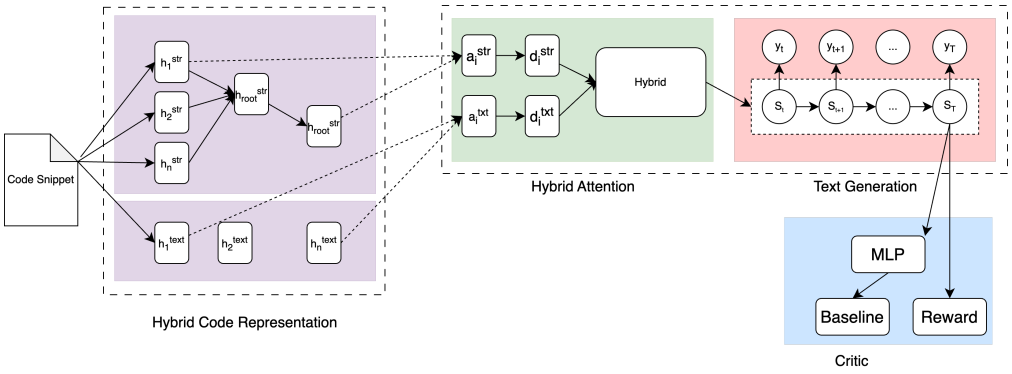


Figure 4.6 Model architecture for CodeSumDRL [4]

To train the model, code is represented in a hybrid manner in two levels: lexical level (code comments) and syntactic level (code ASTs) [4]. An LSTM is used to convert the natural language comments, and a separate Tree-based-RNN is used to represent the code ASTs. The output of these two layers is then used to represent the input code alongside its comment for training the actor and critic.

DEAR

Li et al. [5] proposed DEAR as a novel DL-based approach for APR, in order to address the limitations of existing models that fix only one statement at a time. DEAR targets bugs that require dependent edits across multiple statements and code hunks. To facilitate this,

DEAR uses a hybrid fault localization process that combines Spectrum-Based Fault Localization (SBFL), a fine-tuned BERT model for identifying fix-together relationships among code hunks, and data flow analysis to expand candidate statements into multi-statement buggy hunks. DEAR leverages AST transformations learned from (bug, fix) pairs, allowing it to model fine-grained edits at the sub-tree level. These sub-tree mappings are then used for learning structured code transformations across distributed and dependent repair locations throughout the code.

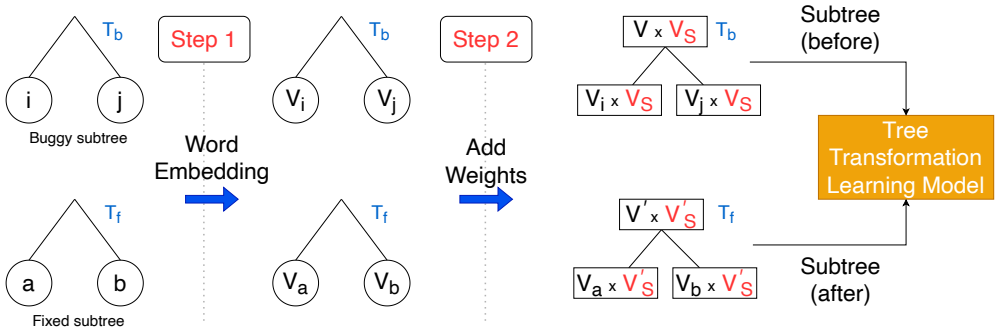


Figure 4.7 Model architecture for DEAR's tree transformation pipeline. V_S and V_S' are summarized vectors learned during the context learning phase [5].

DEAR's architecture uses two tree-based LSTM layers [19], an attention mechanism, and cycle training as presented in Figure 4.7. The first layer encodes each buggy AST sub-tree within the context of the fixed versions of surrounding buggy sub-trees, ensuring that the model learns fixes in a semantically correct manner. The second layer then learns how to map each buggy sub-tree to its fixed counterpart. During training, the ASTs are preprocessed using GloVe embeddings [225] and TreeCaps summarization [226] to create vectorized representations, while CPatMiner [227] is used to extract fine-grained sub-tree mappings between buggy and fixed code.

Recoder

Similar to DEAR, Zhu et al. [6] proposed Recoder, a DL-based approach for APR that introduces a syntax-guided edit decoder to generate edits to the input code rather than generate the entire repaired patch by generating a sequence of AST-level modifications instead of entire repair statements. Since Recoder generates modifications to the ASTs, it addresses the challenge of generating syntactically invalid edits to the input code. Furthermore, by defining edits as structured operations (e.g., insertions or sub-tree modifications) where each operation conforms to the grammar of the programming language, Recoder addresses the

problem of generating small edits. The edit generation process is conditioned on both the faulty code and its context, allowing Recoder to make fine-grained, context-aware decisions at each expansion step. The model architecture builds on a tree-based Transformer [228], which allows for joint encoding of the buggy statement, its method-level context, and the evolving edit structure as shown in Figure 4.8.

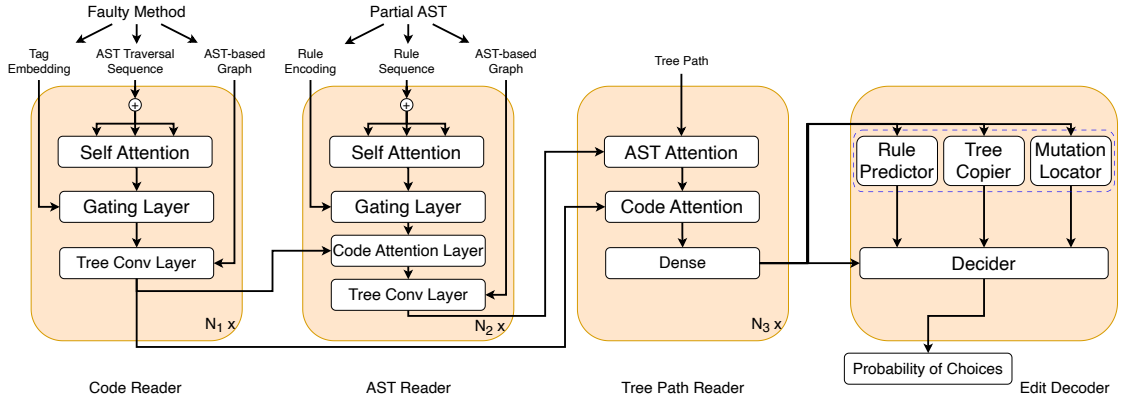


Figure 4.8 Model architecture for Recoder [6]

InferCode

Bui et al. [7] proposed InferCode as a self-supervised approach for learning code representations using ASTs. In order to learn code representations from ASTs, InferCode uses an approach similar to Doc2Vec [229]. Specifically, the AST of a code snippet is treated as a "document" and the sub-trees extracted from the AST (with roots like *expr_stmt*, *decl_stmt*, *expr*, and *condition*) are treated as "words" for prediction. Afterwards, to generate code embeddings, InferCode uses a modified Tree-Based Convolutional Neural Network (TBCNN) [223], by including both token and type information in node embeddings and replacing max-pooling with an attention mechanism for node aggregation. The model then learns to predict the presence of sub-trees given the AST encoding, effectively capturing both syntactic and semantic patterns. Afterwards, the encoder can produce vector representations for any syntactically valid code snippet and be used in downstream tasks such as clone detection, clustering, cross-language code search, and supervised fine-tuning for tasks like code classification and method name prediction.

Type4Py

Mir et al. [8] introduced Type4Py for type inference prediction in Python. Instead of defining type prediction as a classification task over a fixed vocabulary, Type4Py formulates it as a

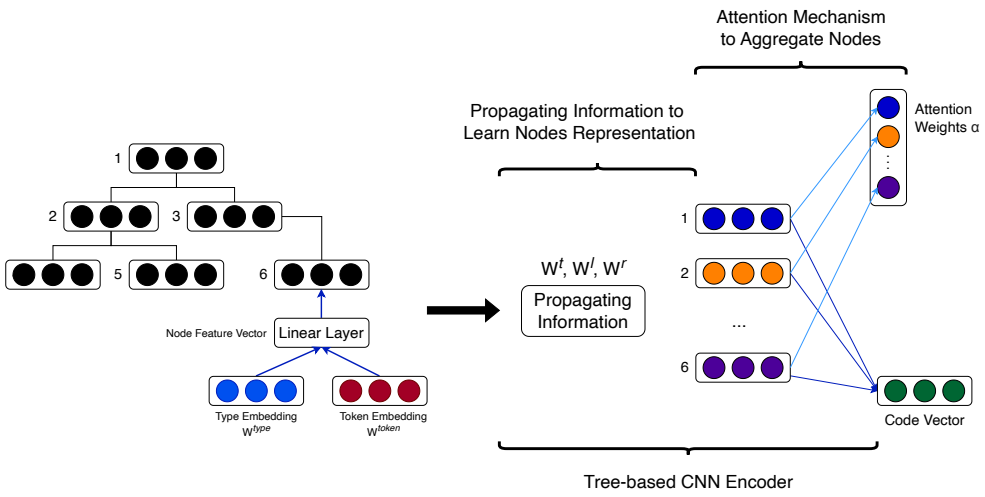


Figure 4.9 Model architecture for InferCode [7]

metric learning problem. Type4py's architecture, as presented in Figure 4.10, consists of two LSTMs. One LSTM separately encodes identifier tokens, and the other encodes code context. The outputs of these layers' vector representations are then concatenated with a sparse binary vector of visible type hints (i.e., user-defined or imported types visible in the module) and passed through a linear layer to produce an embedding in a high-dimensional type space. The model is trained with triplet loss to enforce that embeddings of semantically similar types are close, while dissimilar types are distant. Once the model is trained, for predicting the type of a variable or argument, Type4Py performs a k-nearest neighbor search in the learned type space to predict the input's type.

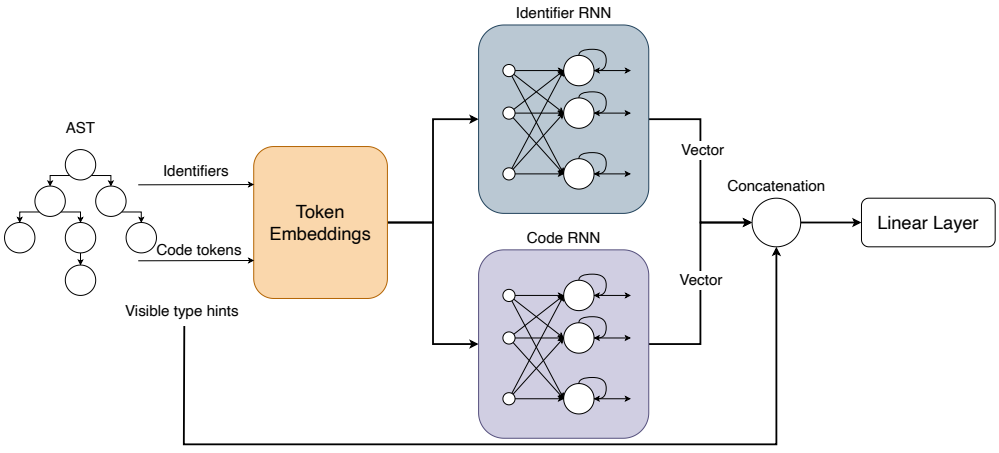


Figure 4.10 Model architecture for Type4Py [8]

For each of the models described above we use the code and data in the replication packages

provided by the authors and train the models from scratch to replicate the results of the original studies if the trained models were not provided [1–8]. It should be noted that for each model, we do not modify the codes or the data in the replication packages, and in cases where it was required (e.g., deprecated libraries, syntax errors in the code, etc.), we kept our modifications as minimal as possible. We provide the source code of each model alongside our probing approach in our replication package [230].

4.3.2 Dataset Construction for Each Model

After training models and replicating the results of the original studies where a trained instance of the model was not available, we use the same data that was used in the original works as input to the models and at the same time extract the AST/CFG from each input and construct the $\langle d, c, u \rangle$ tuple which was explained in Section 4.2.2. Afterward, we extract the intermediate outputs of the model for the given input and train the probe to predict the constructed $\langle d, c, u \rangle$ tuple from the extracted embeddings. It is important to note that the chosen models contain multiple hidden layers. Therefore, for each model, we extract the embeddings from the layers that provide the most information-rich representations learned by the model, as follows:

- AST-NN: The AST-NN model architecture, as shown in Figure 4.3, consists of three main components: an encoding layer, a recurrent layer, and a pooling layer. The encoding layer first converts the input STs as described in Section 4.3.1 into numerical representations suitable for the model. These encoded inputs are then processed by the recurrent layer. The outputs of the recurrent layer are subsequently passed through the pooling layer, which aggregates the representations and produces a fixed-size vector encoding of the entire input STs. For probing, we extract the embeddings from both the recurrent and the pooling layers.

- FuncGNN: The FuncGNN model architecture, as illustrated in Figure 4.4, consists of several components: a graph convolution layer, an attention mask mechanism, and a ReLU activation function. The graph convolution layer learns representations by capturing the structural information from the input CFGs. These learned representations are then passed through the attention mask, which assigns importance weights to different parts of the outputs of the graph convolution layer. Finally, a ReLU activation function is applied to the attention-weighted representations, and a sigmoid activation yields a binary output indicating whether the two input CFGs are code clones or not. For probing, we extract the embeddings from the output of the graph convolution layer.

- SummarizationTF: As displayed in Figure 4.5, the SummarizationTF model follows an encoder-decoder architecture. The encoder component processes the AST generated from a code snippet and generates a contextualized representation from it. The outputs of the encoder layer are used as input to the decoder layer to generate summaries from the input AST. For probing, we extract the embeddings from the outputs of the encoder component.

- CodeSumDRL: The CodeSumDRL model employs a dual-encoder architecture as displayed in Figure 4.6, where separate LSTM layers are used to encode the input code snippet and the associated comments. The outputs of these LSTM layers are then passed as inputs to an encoder layer. For probing, we extract the embeddings from the outputs of the encoder layer.

- DEAR: As displayed in Figure 4.7, the DEAR model is comprised of two tree-based LSTM layers and a fully connected dense layer. In order to generate repairs, the processed AST and its corresponding GloVe code embeddings are used as inputs for the first LSTM layer, the output of which is used as inputs for the second LSTM layer. The outputs of the second LSTM layer are then fed to the dense layer in order to produce the fixed sub-trees for the faulty inputs. For probing, we extract the embeddings from the output of the second LSTM layer.

- Recoder: The Recoder model architecture, as displayed in Figure 4.8, is comprised of four components. In order to generate repairs, the *code reader* component takes the faulty statements and the relative context (methods connected to this statement in the AST) as inputs to encode the information. The outputs of the *code reader* are used as inputs for the *AST reader* component which encodes the partial ASTs that were generated for the edits. The *tree path reader* takes the outputs of both of the previous components as input in order to encode a path from the root node to a non-terminal node. Finally, the outputs of this component are then used as input to the *edit decoder* to produce a probability of each choice for expanding the non-terminal node in the relative context used for inputs to the *code reader*. For probing, we extract the embeddings from the *edit decoder* before the final dense layer.

- InferCode: The InferCode model architecture, as displayed in Figure 4.9, is comprised of a TBCNN layer and a subsequent dense layer. Given that InferCode is proposed as a code encoding approach, for probing, we extract the embeddings from InferCode's outputs directly.

- Type4Py: As displayed in Figure 4.10, Type4Py's model architecture is comprised of two LSTMs and a single fully connected dense layer. Type4Py uses the token embeddings generated from the code's AST as inputs to the identifier LSTM and the code LSTM, the outputs of which are concatenated and then used as inputs for the dense layer for determining the type of the input. Therefore, for probing, we extract the embeddings from the concatenated vector representations generated by the two LSTMs before being used as inputs to the dense layer.

The rationale behind selecting these specific layers is that they are expected to capture the most informative and high-level representations of the input data, as they are the final representations before subsequent transformations or output generations. After extracting the embeddings, the dataset used for training *DeepCodeProbe* for each model consists of (`embeddings`, $\langle d, c, u \rangle$) tuples from the inputs given to each model. We outline the detailed dataset construction for each of the models under study as follows.

AST-NN

Algorithm 4 presents how we construct the $\langle d, c, u \rangle$ tuples for probing AST-NN using *DeepCodeProbe*. Note that to represent the labels of each node, we use the same Word2Vec model that is used by AST-NN to convert the labels into numerical representations (i.e., indexes) as described in Section 4.3.1. We probe this model on its capability to represent the syntax of both C and Java programming languages. We also adopt the same preprocessing approach as AST-NN to convert the inputs from code to STs. Afterward, for each ST, we construct the $\langle d, c, u \rangle$ tuple to be used for training the probe.

FuncGNN

Algorithm 5 presents an overview of the $\langle d, c, u \rangle$ tuple construction for FuncGNN in *DeepCodeProbe*. The training data for FuncGNN is already pre-processed from code into CFGs using Soot[1], a framework that is used to transform Java bytecode into an intermediate representation for analysis, visualization, and optimization. In this format, each node represents a line in the program, and the edges between the nodes indicate the control flow between each node. For constructing the $\langle d, c, u \rangle$ tuple, we traverse the CFG and use the depth, connections, and labels of each node to represent it in a numerical vector for training and evaluating the probe.

[1]https://github.com/Sable/soot

Algorithm 4 Generating $\langle d, c, u \rangle$ Tuples from STs

Require: An Abstract Syntax Tree (AST) *tree*
Ensure: Dictionary containing depth (D), child indices (C), and unary label (U) tuples for ST blocks

procedure TREE2TUPLE(*tree*)
  $word2VecEmbeddings \leftarrow$ Load appropriate Word2Vec model based on language
  $maxTokenIndex \leftarrow$ Number of vectors in $word2VecEmbeddings$
  $vocab \leftarrow word2VecEmbeddings$
  function CONVERTNODETOINDEX(*node*)
    $tokenIndex \leftarrow$ Index of *node.token* in *vocab* or $maxTokenIndex$ if not found
    $childIndices \leftarrow$ Empty list
    for *child* in *node.children* **do**
      $childIndices$.append(CONVERTNODETOINDEX(*child*))
    end for
    return $[tokenIndex] + childIndices$
  end function
  function TREE2INDEX(*rootNode*)
    blocks $\leftarrow$ Extract ST blocks from AST starting from *rootNode*
    $sequenceInfo \leftarrow$ Initialize empty dictionary for block info
    for $i = 0$ to $length(blocks) - 1$ **do**
      $block \leftarrow blocks[i]$
      $blockTokenIndex \leftarrow$ Index of *block.token* in *vocab* or $maxTokenIndex$
      $blockChildrenIndices \leftarrow$ CONVERTNODETOINDEX(*block*)
      $sequenceInfo[i].d \leftarrow i + 1$
      $sequenceInfo[i].c \leftarrow$ Flatten $blockChildrenIndices$ excluding the first element
      $sequenceInfo[i].u \leftarrow blockTokenIndex$
    end for
    $D \leftarrow$ Extract d values from $sequenceInfo$ for all blocks
    $C \leftarrow$ Extract c values from $sequenceInfo$ for all blocks
    $U \leftarrow$ Extract u values from $sequenceInfo$ for all blocks
    return $\{d : D, c : C, u : U\}$
  end function
  return TREE2INDEX(*tree.root*)
end procedure

Algorithm 5 Generating $\langle d, c, u \rangle$ Tuples from CFGs

Require: Code graphs represented as lists of edges and node labels
Ensure: Tuples representing depth (D), connections (C), and unary label (U) for the code graphs

function CFG2TUPLE($codeGraphs, nodeLabelMappings$)
  for $codeGraph$ in $codeGraphs$ **do**
    $edges \leftarrow codeGraph.edges$
    $reverseEdges \leftarrow$ list of edges with source and target swapped
    $edgeList \leftarrow edges + reverseEdges$ ▷ Ensure connections in both directions
    Convert $edgeList$ to an appropriate data structure
    $depths \leftarrow$ list of node depths in the graph traversal order
    $connections \leftarrow edgeList$
    $usages \leftarrow$ list of node label mappings from $nodeLabelMappings$
    Store $(depths, connections, usages)$ as a tuple for the current $codeGraph$
  end for
  return list of $(depths, connections, usages)$ tuples for all code graphs
end function

SummarizationTF

This model works with ASTs. Therefore, unlike AST-NN there is no need to first generate the ASTs from code and then break them down into STs. As such, the process for generating the $\langle d, c, u \rangle$ tuples is similar to that of Algorithm 4 with some modifications to the method of parsing the trees. Algorithm 6 presents our approach in detail.

CodeSumDRL

Similar to SummarizationTF, CodeSumDRL also works with ASTs generated from code. As such, we follow the same steps as outlined in Algorithm 6 to build the necessary $\langle d, c, u \rangle$ tuples for training and evaluating the probe.

DEAR

DEAR accepts ASTs as input. However, similar to AST-NN, it further breaks down ASTs into sub-trees using CPatMiner for pre-processing. As such, to construct the $\langle d, c, u \rangle$ tuples for DEAR, similar to AST-NN, we follow the same steps in Algorithm 4 to derive the $\langle d, c, u \rangle$tuples from these sub-trees, with the difference of using DEAR's own GloVe embeddings instead of the Word2Vec model for extracting the u tuples.

Algorithm 6 Generating $\langle d, c, u \rangle$ Tuples from ASTs

Require: Source code *code*
Ensure: Tuples of node positions (D), children count (C), and unary label (U) for AST processing

procedure TREE2TUPLE(AST)
    $token2Index \leftarrow$ Load appropriate tokenizer based on the programming language
    $maxTokenIndex \leftarrow$ Highest index in $token2Index$
    $vocab \leftarrow token2Index$
    function CONVERTNODETOINDEX($node$)
        $tokenIndex \leftarrow$ Index of $node.token$ in $vocab$ or $maxTokenIndex$ if not found
        $childIndices \leftarrow$ Empty list
        for *child* in $node.children$ **do**
            $childIndices$.append(CONVERTNODETOINDEX($child$))
        end for
        return $[tokenIndex] + childIndices$
    end function
    function AST2INDEX($tree$)
        $nodes \leftarrow$ Extract nodes from $tree$
        $sequenceInfo \leftarrow$ Initialize empty dictionary for node info
        for $i = 0$ to $length(nodes) - 1$ **do**
            $node \leftarrow nodes[i]$
            $nodeTokenIndex \leftarrow$ Index of $node.type$ in $vocab$ or $maxTokenIndex$
            $nodeChildrenIndices \leftarrow$ CONVERTNODETOINDEX($node$)
            $sequenceInfo[i].d \leftarrow$ Extract positions for $node$ in AST
            $sequenceInfo[i].c \leftarrow nodeChildrenIndices$
            $sequenceInfo[i].u \leftarrow nodeTokenIndex$
        end for
        $D \leftarrow$ Extract position values from $sequenceInfo$ for all nodes
        $C \leftarrow$ Extract children count from $sequenceInfo$ for all nodes
        $U \leftarrow$ Extract type indices from $sequenceInfo$ for all nodes
        return $\{d : D, c : C, u : U\}$
    end function
    return AST2INDEX(AST)
end procedure

Recoder

Recoder works on ASTs generated from codes as input. As such, we follow the steps as presented in Algorithm 6 to construct and extract the necessary $\langle d, c, u \rangle$ tuples used for training and evaluating the Probe.

InferCode

As detailed in Section 4.3.1, similar to AST-NN, InferCode is proposed as a code representation learning approach with ASTs as inputs. However, in contrast to AST-NN, it works on full ASTs. Therefore, for constructing the dataset, we follow the same steps presented in Algorithm 6.

Type4Py

Similar to InferCode, Type4Py also works on full ASTs. As such, we use the same steps presented in Algorithm 6 to construct and extract the $\langle d, c, u \rangle$ tuples for the probe's training and evaluation dataset.

As detailed in Section 4.2.2 and presented in Algorithm 2, our proposed approach for constructing the $\langle d, c, u \rangle$ tuples is generic and extendable to any model that is trained on syntactically valid representations of code. As shown in the details of dataset construction for each of the models under study above, the d and c tuples are directly extracted from the AST/CFG of the input code itself, and the u tuple is extracted from the model's embedding/tokenization step during pre-processing. This makes *DeepCodeProbe* extendable to other syntactically valid representations and models.

4.3.3 Probe's Design for Each Model

As explained in Chapter 2.4, the probe should have minimal size and complexity. Following the same conventions used in probing language models [231], the probe should not learn any information from the data but instead, learn a transformation (i.e., projection) of the embeddings that exposes the information that the model learns from its training data. Therefore, for each of the models under study, based on their capacity (i.e., number of parameters), we experiment with probes of different sizes. To prevent the probes from learning representations from the model's embeddings, we follow the same design approach as used in [212]. This means that each probe has a single hidden layer, and the number of units in this layer is adjusted based on the size and architecture of the model being probed. To determine the

number of units in the probe for each model, we use a grid search strategy with the constraint that the number of units must not exceed the encoding size of the model under study (i.e., if the model's encoding layer produces embeddings with 200 features, the probe's number of units must be smaller than 200). This constraint is necessary because larger probes would have higher capacity than the model's own encoder, potentially leading the probe to memorize or overfit instead of projecting the model's learned representations [232]. The optimal size is selected based on the probe's performance: probes that are too small fail to converge, while those that are too large tend to overfit.

Probes are trained using the Adam optimizer [233] with an initial learning rate of 0.001. If the validation loss fails to improve for 5 consecutive epochs, the learning rate is reduced by a factor of 10 to facilitate convergence. Early stopping with a patience of 5 epochs is applied, and the batch size for all experiments is set to 32. Probe training continues until performance on the evaluation set fails to improve for 5 epochs or early stopping is triggered. Using different probe sizes for each model is essential due to architectural and capacity differences across models, as detailed in Section 4.3.1. Similarly, not fixing the number of training epochs is important given the varying convergence speeds and dataset sizes.

Table 4.1 Probe hyperparameter search space for each model

Model	**Model Hidden Dim**	**Probe Rank**	**Probe Hidden Dim Range**	**Hidden Dim Step Size**
AST-NN	100	100	{32–100}	32
FuncGNN	512	512	{64–512}	64
SummarizationTF	512	512	{64–512}	64
CodeSumDRL	512	512	{64–512}	64
DEAR	128	128	{32–128}	32
Recoder	1024	1024	{128–1024}	128
InferCode	100	100	{32-100}	32
Type4Py	512	512	{64–512}	64

Table 4.1 presents the hyperparameter search space for each model. The "Model Hidden Dim" column indicates the dimensionality of the original model's learned representations. The "Probe Rank" column specifies the size of the probe's input projection space, which corresponds to the extracted representation dimension. The "Probe Hidden Dim Range" column contains the range of hidden dimensions explored during the grid search, and the "Hidden Dim Step Size" column indicates the increment used during the sweep. We include the code for training and evaluating the probes, alongside the full hyperparameters used for each of the models under study, in our replication package [230].

4.3.4 Scaling Up the Models

Kaplan et al. [234] examined the scaling laws for language models by investigating how increasing model capacity, training data size, and training computational resources impact training efficiency and performance. In their study, they experiment with how increasing each and a combination of these factors affects the model's sample efficiency, overfitting, convergence, and overall performance on the test benchmarks. Hoffmann et al. [235] extend their findings by highlighting a tradeoff between model capacity and dataset size under fixed compute, showing that increasing capacity without sufficient data can harm generalization. Furthermore, Lin et al. [236] report similar trends in models trained on code tokens with fewer than 100 million parameters, validating that these effects persist even in smaller-scale settings. Although these works focus on transformer-based models trained on code tokens, we can apply the same principles to the models under study. As such, based on these scaling principles, to address RQa_3, we keep the amount of compute and the dataset size the same, while increasing the number of models' parameters (i.e., capacity) without changing the models' architecture. We have described the architecture of each of the models under study alongside the layers from which we have extracted the embeddings in Sections 4.3.1 and 4.3.2, respectively. To align our approach with the principles outlined by Kaplan et al. [234], we define "scaling up" for each model in our study as increasing model capacity by expanding the number of units within the layers responsible for generating outputs used for probing. Specifically, our scaling strategy includes:

- AST-NN: Increasing the recurrent and pooling layers' unit count and leaving the encoding layer unchanged.
- FuncGNN: We increase the number of units of the graph convolution layer without altering the number of attention heads.
- SummarizationTF: The Multi-way Tree-LSTM's unit count is increased, while all other components remain the same.
- CodeSumDRL: The encoder's unit count is increased, with no modifications to the rest of the components.
- DEAR: The tree-based LSTM layers' unit count is increased while leaving the dense layer unchanged.
- InferCode: Increasing the convolution layer's size without altering the subsequent dense layer.

- Type4Py: Increasing both LSTM layers' unit counts and leaving the dense layer unchanged.

As described above, instead of increasing the number of units in all model components, we only target the specific layers responsible for generating representations from the inputs. This targeted scaling allows us to isolate the impact of increasing models' representational capacity on their ability to retain syntactic information. It must also be noted that even a single increase in the size of one layer can lead to a significant increase in the total number of the model's learnable parameters. For example, for DEAR, increasing the hidden dimension of the LSTM layers from 32 to 64 raises each layer's learnable parameters from 12,416 to 33,024, more than doubling each layer's capacity. As such, our focused scaling strategy is both efficient and necessary for studying syntax learning behavior, given the effects of scaling up each layer's number of units. Furthermore, as explained in Section 4.3.1, Recoder is a transformer-based model with already a large number of parameters (more than 90 million). Given its results in retaining syntactic information, as we will explore in detail in Section 4.4.1, and given that our study is focused on the effects of scale on small DL models, we do not scale Recoder.

Finally, we do not scale models indefinitely. Instead, we increase their capacity step by step and retrain each model until we observe clear signs of diminishing returns (i.e., validation loss increases, or model performance degrades from the reported metrics in the original works). At that point, further scaling is no longer required, as the model begins to diverge from its original training objectives. This approach is consistent with prior work that investigates the boundaries of model scaling under resource constraints. Studies by Hestness et al. [237], Rosenfeld et al. [238], and Hernandez et al. [239] show that performance improves with model size up to a threshold, after which further increases lead to overfitting or diminishing gains without further increasing the amount of training data. Given that we keep the dataset size fixed, probing such degraded models for syntactic information would be unreliable, as their internal representations no longer reflect successful learning, and subsequent $\langle d, c, u \rangle$ representations would be invalid.

4.4 Results and Analysis

Before analyzing the embeddings extracted from the models to assess their capability to learn the syntax of the programming language and answer our research question, we must first ensure the reliability of the probing approach used to obtain those embeddings. Therefore, to answer RQa_1, we first conduct a validation of our probing approach in Section 4.4.1. Next,

we will present the results of our analyses for RQa_2 in Section 4.4.2 and RQa_3 in Section 4.4.3.

4.4.1 RQa_1: Assessment of Code Representations on Preserving Syntactic Properties

As described in Section 4.2.2, we evaluate *DeepCodeProbe* by first assessing the validity of the $\langle d, c, u \rangle$ tuple and the embeddings extracted from the models. As AST-NN and FuncGNN are designed specifically for CCD, we use the same dataset provided by their authors (AST-NN [1] and FuncGNN [2]) for the validation of *DeepCodeProbe* on these models. For the rest of the models under study, the authors do not provide a CCD dataset, as these models were proposed for different tasks. Therefore, to assess the validity of our probing approach, we use datasets that have been used throughout the literature as CCD benchmarks, which we detail below.

Table 4.2 presents the details of the datasets used for the validation of our data representation approach. We use the OJClone dataset provided in [223] for validation of our $\langle d, c, u \rangle$ tuples derived from C codes. It should be noted that the original dataset is mainly composed of clones of Types 3 and 4. Given that we aim to validate the representation approach as described in Section 4.2.2, we only select clones of Types 1 and 2 that have similar ASTs. Similarly, we use the BigCloneBench dataset provided by [222] for our validation process on Java. For models that are trained on Python codes, we use the CodeSearchNet dataset provided in [240]. It should be noted that, unlike the C and Java datasets, this dataset does not provide a fine-grained classification of clone types and instead only contains pairs of codes that are clones of each other and pairs that are not.

Table 4.2 Breakdown of the datasets used for validation of $\langle d, c, u \rangle$ tuples

Dataset	Programming Language	Similarity Type	Number of Samples	Ratio of Samples (%)
OJClone	C	Similar	3286	39.11
		Dissimilar	5114	60.87
BigCloneBench	Java	Similar	20809	57.23
		Dissimilar	15554	42.77
CodeSearchNet	Python	Similar	2694311	50.0
		Dissimilar	2694311	50.0

Table 4.3 presents the average cosine similarity for the $\langle d, c, u \rangle$ tuples for each of the models under study. The "Programming Language" column specifies the programming language on which the model is trained. The "Comparison criterion" denotes which subsets of the datasets were used for the similarity comparison, with "similar" indicating comparison among code clones and "dissimilar" indicating comparison among non-clone pairs. Finally, the "Average

Similarity" columns show the average cosine similarity across all $\langle d, c, u \rangle$ tuples constructed for each model and comparison criterion.

Table 4.3 Results of cosine similarity of $\langle d, c, u \rangle$ tuples for each model

Model	**Programming Language**	**Similarity Criterion**	**Average Similarity D (%)**	**Average Similarity C (%)**	**Average Similarity U (%)**
AST-NN	C	Similar	64	17	43
		Dissimilar	57	12	4
	Java	Similar	80	50	76
		Dissimilar	42	16	54
FuncGNN	Java	Similar	90	86	82
		Dissimilar	57	36	68
SummarizationTF	Java	Similar	75	83	74
		Dissimilar	33	60	51
CodeSumDRL	Python	Similar	0	28	35
		Dissimilar	-11	3	20
Dear	Java	Similar	57	11	39
		Dissimilar	26	2	7
Recoder	Java	Similar	75	83	76
		Dissimilar	33	60	48
InferCode	Java	Similar	75	83	67
		Dissimilar	33	60	23
Type4Py	Python	Similar	0	28	28
		Dissimilar	-11	3	13

As shown by the results in Table 4.3, we can observe a significant difference in the $\langle d, c, u \rangle$ similarities between clone and non-clone code pairs across all models. Specifically, for AST-NN we can observe a 7% difference in the representations of d tuples which encode node location, a 5% difference in the representation of c tuples which encode the node's children, and a major difference of 39% between u tuples which encode node labeling information which lines up with how large ASTs are broken down into smaller STs for this model [1]. For FuncGNN, we observe noticeable differences in representations constructed for CFGs, showing a strong differentiation based on node position, connections, and labels between clone and non-clone pairs especially across the d tuples which encode the position of each node in the CFG and the c tuples which encode the connections between the nodes. We observe a similar trend for SummarizationTF, especially with a difference of 43% for the d tuple, 23% for the c tuple, and 23% for the u tuple which lines up with how ASTs of dissimilar codes

have different structures as also described in Section 4.2.2. For CodeSumDRL, we observe similar results as SummarizationTF in the similarities between the $\langle d, c, u \rangle$ tuples as well as a similar difference of distinction between the encoded representation of the nodes' positions and children. For DEAR, we observe a 31% difference between the d tuples, 9% between the c tuples, and 32% for the u tuples. As described in Section 4.3.1, similar to AST-NN, DEAR breaks ASTs down into smaller sub-trees. The differences between these sub-trees are also observed in our d and c tuples, with the high differences between the d tuples, which encode each node's positional information, and lower differences between the c tuples that encode the node's children information. The high difference observed in the u tuples also reflects how DEAR encodes AST node labels using pre-trained GLoVe embeddings [5]. As detailed in Section 4.2.2, both d and c tuples are derived directly from the ASTs. Given that we use the same dataset to validate the data representations, and since Recoder and InferCode work on full ASTs, their average similarity values for the d and c tuples are the same as those observed for SummarizationTF. However, Recoder shows a 28% difference between the u tuples of similar and dissimilar codes, while InferCode shows a 44% difference between the same samples. Finally, for Type4Py, we observe identical similarity values for the d and c tuples to those of CodeSumDRL, given the same underlying dataset, with a 15% difference in the u tuples between similar and dissimilar codes.

Based on these results, we can infer that our $\langle d, c, u \rangle$ tuple representation, extracted from ASTs/CFGs, holds sufficient syntactical information and can be used for probing for syntactic information on the models under study.

With the validity of the constructed $\langle d, c, u \rangle$ tuples established, we proceed to assess the validity of the information within the embeddings extracted from each model as described in Section 4.2.2. Table 4.4 presents the results of embedding validation. Following a similar methodology to validating the $\langle d, c, u \rangle$ tuples representation, we compute the average cosine similarity across both clone and non-clone pairs. Additionally, the "Trained" column indicates whether the embeddings were extracted from an untrained or trained instance of the model.

The analysis of data from Table 4.4 reveals a significant distinction between the embeddings of clone and non-clone codes, regardless of the objective for which the models were trained. Specifically, we observe that after training the models, the similarity of the extracted embeddings for code clones increases. Furthermore, the distance in the embeddings extracted from the models between clone and non-clone codes changes significantly post-training, as opposed to their randomly initialized stage. In more detail, for code clones, AST-NN shows a significant increase from 4.81% to 28.89% in C, and from 0.42% to 36.43% in Java, post-training. We observe an increase in the cosine similarity of non-clone codes as well, but

Table 4.4 Results of cosine Similarity of embeddings before and after training

Model	Programming Language	Similarity Criterion	Trained	Average Cosine Similarity (%)
AST-NN	C	Similar	Y	28.89
			N	4.81
		Dissimilar	Y	20.38
			N	1.90
	Java	Similar	Y	36.43
			N	0.42
		Dissimilar	Y	20.12
			N	0.51
FuncGNN	Java	Similar	Y	86.15
			N	78.46
		Dissimilar	Y	72.55
			N	71.05
SummarizationTF	Java	Similar	Y	59.33
			N	52.90
		Dissimilar	Y	53.47
			N	43.97
CodeSumDRL	Python	Similar	Y	19.03
			N	8.71
		Dissimilar	Y	11.60
			N	3.19
Dear	Java	Similar	Y	30.66
			N	12.34
		Dissimilar	Y	15.66
			N	3.74
Recoder	Java	Similar	Y	67.37
			N	33.33
		Dissimilar	Y	21.82
			N	25.03
InferCode	Java	Similar	Y	60.66
			N	38.50
		Dissimilar	Y	23.33
			N	21.50
Type4Py	Python	Similar	Y	37.40
			N	10.09
		Dissimilar	Y	15.33
			N	9.30

to a lesser degree, which can be attributed to the model's ability to identify some form of structure in the STs which we expand more on, in RQa_2 (Section 4.4.2). We observe a similar pattern for FuncGNN, with the similarity of code clones increasing from 78.46% to 86.15%, and just a 1.5% increase between the embedding similarity of non-clone codes, post-training. The marginal increase for non-clone codes shows the model's capability to recognize underlying structural patterns, which we will expand on when discussing the probing results in Section 4.4.2. For SummarizationTF, we also observe the same pattern of average cosine similarity increasing for code clones after training the model with a lower cosine similarity for non-clone pairs. We observe a 6.4% difference between the embeddings of code clones before and after training the model and a 9.5% increase for non-clone codes. For CodeSumDR,L we observe that the embedding similarity increases from 8.71% to 19.03% for code clones, and from 3.19% to 11.6% for non-clones. DEAR displays an embedding similarity of 30.66% for clone pairs and a 15.66% for non-clone pairs once the model has been trained. We can observe a significant difference between the embedding similarities of clones and non-clones pre-training as well. We attribute this to how DEAR also breaks down the full ASTs to multiple subtrees similar to AST-NN. Recoder displays a significant difference of 45.55% between the similarity of embeddings for clones (67.37%) and non-clones (21.82%) post model training. InferCode also shows a significantly high difference of 37.33% between the similarity of the embeddings of clones (60.66%) and non-clones (23.33%). This is expected as InferCode is trained on ASTs and is a code representation approach. Furthermore, we observe a difference of 17% between the embedding similarities of clones and non-clones before InferCode is trained. We attribute this difference between embeddings of the randomly initialized model to the use of TBCNNs, as convolutional architectures are known to exhibit strong inductive biases toward local pattern matching even when randomly initialized [241]. Finally, for Typ4Py, we observe that the embedding similarity increases from 10.09% to 37.40% for code clones and from 9.30% to 15.33% for non-clones after the model is trained.

These results confirm that the extracted embeddings are suitable for probing, as they reflect meaningful syntactic distinctions between clone and non-clone code pairs across all models. For each model under study, we observe consistent and significant differences in embedding similarity between clones and non-clones. Since these models are trained on syntactic code representations (ASTs/CFGs), this suggests they retain structural information independent of their primary training objectives (such as CCD, APR, etc.).

By cross-analyzing the data in Tables 4.3 and 4.4, we can infer the following:

- The $\langle d, c, u \rangle$ tuple, which is constructed for each model following the model's data representation approach, successfully encodes the syntactic information in the AST/CFG.

- All models demonstrate an ability to learn some form of syntactic representations (albeit to different degrees) as observed by the differences in embedding similarities between clone and non-clone code pairs, even when clone detection is not part of their original training objective.

Now, with the validity of our approach being established, in the next section, we discuss the results of probing.

4.4.2 $\mathbf{RQa_2}$: Effects of Representation Choices on Quality of Features and Task-specific Requirements

Table 4.5 presents the results of our probing as described in Sections 4.3.2 and 4.3.3. The "Accuracy-D", "Accuracy-C", and "Accuracy-U" columns indicate the probe's accuracy in predicting the d, c, and u tuples, respectively. As we can observe from the results, all the models under study, except for Recoder, are incapable of representing the full syntax of the programming language in their latent space. For AST-NN trained for CCD on C and Java, the poor accuracy scores of 8% across all syntax tuples (d, c, and u) indicate an extremely limited ability to encode the programming language's syntax in its latent space. Similarly, SummarizationTF struggles with syntax learning as well, achieving low accuracies of 13.83%, 14.79%, and 14.79% for the d, c, and u tuples, respectively. In contrast, the CodeSumDRL model exhibits a relatively stronger syntax learning ability, with accuracies of 41.6% for d, 33.92% for c, and 28.08% for u tuples on the code summarization task. While not close to the results reported for LLMs in [156, 157, 212] (where the probes exhibit over 80% accuracy in retrieving information related to the programming language's syntax), these higher scores suggest that CodeSumDRL learns some syntactic patterns from code. We observe an accuracy of 37.40%, 13.37%, and 41.92% for recovering the $\langle d, c, u\rangle$ tuples from DEAR's embeddings. The relatively high accuracies of the d and u tuples suggest that while DEAR is incapable of representing the full syntax of the programming language, it is capable of retaining general information about the AST structures of code in Java. We observe that InferCode's probe achieves an accuracy of 12.09% for recovering the d tuple, 41.33% for recovering the c tuple, and a relatively high accuracy of 66.66% for recovering the u tuple, which encodes the AST node types' information. Finally, Type4Py's probe achieves accuracies of 24.58%, 39.04%, and 61.20% on predicting the d, c, and u tuples, respectively. Similar to InferCode, we observe that Type4Py also shows a relatively high accuracy on predicting the type of a node in the input code's AST.

The most intriguing results come from FuncGNN and Recoder probes. As described in Section 4.3.1, FuncGNN was designed for CCD on CFGs. Remarkably, it achieves 98% accuracy

Table 4.5 Result of *DeepCodeProbe*'s accuracy on recovering syntactic information

Model	Task	Programming Language	Accuracy D(%)	Accuracy C(%)	Accuracy U(%)
AST-NN	CCD	C	8.65	8.63	8.65
		Java	8.33	8.09	8.65
FuncGNN	CCD	Java	43.36	98.51	39.26
SummarizationTF	Code Summarization	Java	13.83	14.79	14.79
CodeSumDRL	Code Summarization	Python	41.60	33.92	28.08
Dear	APR	Java	37.40	13.37	41.92
Recoder	APR	Java	78.33	73.50	82.33
InferCode	Code Representation	Java	12.09	41.33	66.66
Type4Py	Type Inference	Python	24.58	39.04	61.20

in predicting the c tuple that encodes edge connection information in the code's CFG. Thus, we can conclude that FuncGNN particularly focuses on the edges of the CFGs (connections between nodes) in order to detect clones between two input codes. This observation also aligns with the results reported in [2], in which the goal of FuncGNN is to predict the GED between the CFGs of two input codes. However, its performance of 43.36% and 39.26% for d and u tuples indicates that the model is incapable of representing the full syntax of the programming language. On the other hand, Recoder demonstrates the highest syntax learning capability among all models under study, with consistently high accuracies across all three tuple types: 78.33% on the d tuples, 73.50% on the c tuples, and 82.33% on the u tuples. While some of the models under study exhibit relatively high accuracy on individual tuple types, such as FuncGNN's 98.51% accuracy on the c tuples or InferCode's 66.66% accuracy on the u tuples, none of them maintain high performance across all three tuples simultaneously. Recoder is the only model to achieve this level of comprehensive syntactic representation, indicating a strong ability to capture both structural and token-level syntax in Java code.

From the results of our probing, we can see that our probes are not capable of predicting the $\langle d, c, u\rangle$ tuples for their input codes for AST-NN and SummarizationTF which, as described in Section 4.2.2, indicates that they are incapable of learning the syntax of the programming language that they are trained on. Even though in comparison to the mentioned models, CodeSumDRL's probe achieves higher accuracies across the $\langle d, c, u\rangle$ tuples, the results show that its syntax learning capability is limited. Additionally, the cosine similarity analysis in Table 4.4 reveals a significant difference in the quality of embeddings before and after train-

ing. While models like AST-NN and CodeSumDRL show improved embedding alignment after training, FuncGNN and SummarizationTF exhibit less consistent behavior, indicating variability in how they represent syntactic information. As mentioned above, Recoder achieves the highest performance in predicting the $\langle d, c, u \rangle$ tuples for their input codes. These results show that Recoder specifically focuses on the position of each node and its labeling information for APR. These results also line up with those reported in [6] and how Recoder is designed to produce repairs to programs by modifying the faulty nodes in the input ASTs. On the other hand, probing InferCode reveals interesting insights into what it learns and how it uses the learned patterns from the ASTs to encode a code snippet into a vector representation. Specifically, we can observe that InferCode is not capable of predicting the position of a node in the AST, however, it is capable of retaining distinctive information about the structure of the tree as indicated by the relative high accuracy the c tuples and the types of nodes in the tree similarly indicated by the high accuracy of the u tuples. Furthermore, Type4Py's probe achieves an accuracy of 61.20% for encoding the labeling information of a node in the AST. This lines up with how Type4Py is proposed as an approach for predicting the type of a variable given the AST of the code as input.

Despite these limitations in syntactic learning, all models still perform well on their respective benchmarks, suggesting that they may be learning abstract patterns from their data representations to achieve their tasks. This raises an important question: if these models are not explicitly capturing syntax, what types of representations are they leveraging to perform effectively?

Findings 1: By analyzing the embeddings extracted from the models under study for clone and non-clone pairs, we can observe that the models learn some information about the programming language's syntax. However, our probe shows that none of the models under study are capable of representing the complete syntax in their latent space. This indicates that the models under study are either incapable of representing the full syntax of the programming language in their latent space due to their limited capacity (as opposed to LLMs) or do not require the full syntax to achieve their objectives.

Our findings in Table 4.5 indicate that while the models under study struggle to capture the complete syntax of the programming language, they still retain some degree of syntactic information due to being trained on syntactically valid representations. The cosine similarity results between clone and non-clone pairs in Table 4.3 show that the extracted embeddings from the models under study exhibit some retention of structural patterns. This suggests that while full syntax learning is limited, these models are still capable of capturing abstract

representations of syntactic features. Therefore, we aim to investigate these syntactic abstractions. Furthermore, as our results for FuncGNN reveal what it learns and which parts of the CFGs it pays attention to for CCD, we do not further analyze it here. As such, to investigate the representations learned by the models, we keep the probing approach the same while changing the $\langle d, c, u \rangle$ tuple construction as follows, to probe the models for abstractions of syntactic information:

- c: A binary flag for each node that indicates whether the node has a child in the AST or not. Here, as opposed to the approach taken in RQ1, we do not include the child information for each node to be reconstructed and instead only investigate whether the models under study are capable of learning whether a node in the AST has children or not.
- u: The direct label of the node extracted from the AST. As opposed to the original u tuple, which was extracted from the vector representation used by each model.
- d: We have decided to discard this vector representation since the new C and U tuples are insufficient for reconstructing the AST. As such, encoding the position information is no longer helpful.

Algorithm 7 details the process of generating the $\langle c, u \rangle$ tuples from an AST, with Listing 2 showing the $\langle c, u \rangle$ tuple constructed from the running example presented in Figure 4.2. Although the $\langle c, u \rangle$ tuples are not syntactically valid, they are direct abstractions of the $\langle d, c, u \rangle$ tuples. As such, we expect them to exhibit similar patterns in the validation results, albeit with different values. Table 4.6 presents the results of validating the $\langle c, u \rangle$ tuples using the same approach previously applied to the $\langle d, c, u \rangle$ tuples, as described in Section 4.2.2.

From Table 4.6, we observe that for all models under study, the $\langle c, u \rangle$ tuples for both similar and dissimilar code show higher cosine similarities compared to the results presented in Table 4.3. Given that we no longer encode the direct structure of the ASTs in the c tuple and instead use raw labels from the ASTs for the u tuple, rather than relying on each model's own embedding scheme, such higher values are expected.

Table 4.7 presents the results of our probing using the new $\langle c, u \rangle$ tuples. We observe only slight improvements for CodeSumDRL across both c and u tuples. For Type4Py, the c tuple accuracy increases by just 3.46%, while the u tuple accuracy improves by 17.51%. In contrast, the remaining models show significant improvements in accurately recovering these abstract representations. These findings highlight the importance of selecting a task-specific data representation that aligns with a model's architecture and training objective, as

Algorithm 7 Generating $\langle c, u \rangle$ Tuples from ASTs

Require: Source code *code*
Ensure: Tuples of binary children presence (C) and direct node label (U) for AST abstraction probing

```
1: procedure TREE2CUTUPLE(AST)
2:     function HASCHILDREN(node)
3:         return 1 if node.children ≠ ∅ else 0
4:     end function
5:     function AST2CUTUPLE(tree)
6:         nodes ← Extract nodes from tree
7:         C ← Empty list
8:         U ← Empty list
9:         for node in nodes do
10:            hasChild ← HASCHILDREN(node)
11:            nodeLabel ← Label of node.type in AST
12:            C.append(hasChild)
13:            U.append(nodeLabel)
14:        end for
15:        return {c : C, u : U}
16:    end function
17:    return AST2CUTUPLE(AST)
18: end procedure
```

it suggests that models do not need to fully learn the programming language's syntax, and an appropriate abstraction can be sufficient.

Conversely, the slight enhancement observed in CodeSumDRL's probe results can be attributed to two factors:

- Even though CodeSumDRL uses ASTs to represent code, in contrast to AST-NN and SummarizationTF, it follows a multi-step process to encode information (an LSTM for processing the natural language descriptions and a Tree-based-RNN for processing the AST) as described in Section 4.3.1. As such, we will further investigate CodeSumDRL's syntax learning capabilities in RQa_3 (Section 4.4.3).
- The complex data representation scheme, which abstracts many of the details from the AST alongside the limited capacity of the model, results in a loss of syntactic information.

Despite CodeSumDRL's limited capacity to represent detailed and abstract syntactic information, it achieves commendable results on established code summarization benchmarks [4]. The performance of CodeSumDRL on code summarization benchmarks alongside the high level of abstract syntax representation retention on AST-NN and SummarizationTF indicates

```
c = [
        1 # node 1 has children.
        1 # node 2 has children,
        1 # node 3 has children,
        .......
        0 # node 8 has no children
        0 # node 9 has no children
        .....
    ]
u = ['Module', 'FunctionDef', 'Expr', 'Arguments', ....]
```

Listing 2: $\langle c, u \rangle$ tuple example for AST probing.

the importance of using explicit information encoding for small models. This observation is particularly relevant given our choice of models to study. As described in Section 4.3.1, we chose models that are trained not on raw code (which contains syntax information implicitly) but on the syntactically valid representation of code (which contains the syntax explicitly). By cross-analyzing the results of tables 4.3, 4.4, 4.5, and 4.7 we can infer the following:

- For smaller ML models, using representations that explicitly encode the syntax of the programming language allows for some level of syntax learning capability regardless of the task and models' capacity.
- AST and CFG representations of code exclude redundant details such as comments, docstrings, and variable names, resulting in a significantly reduced representation space compared to using raw code text as input for models. This streamlined representation space is advantageous as it allows models to learn abstractions of the programming language's syntax even when the models are not large (i.e., have a high number of learnable parameters).

Findings 2: By using a more abstract data representation for probing the models, we can observe a significant increase in the syntactic information captured in their latent space, except for CodeSumDRL, which uses complex representations of ASTs. For the rest of the models under study, we can observe that they are capable of retaining information about structural information and the relationship between nodes in an AST to accomplish their tasks without the need to learn the entire syntax of the programming language.

Table 4.6 Results of cosine similarity of $\langle c, u \rangle$ tuples for each model

Model	Programming Language	Similarity Criterion	Average Similarity C (%)	Average Similarity U (%)
AST-NN	C	Similar	44	60
		Dissimilar	32	27
	Java	Similar	59	80
		Dissimilar	22	34
SummarizationTF	Java	Similar	71	88
		Dissimilar	39	53
CodeSumDRL	Python	Similar	62	49
		Dissimilar	16	21
Dear	Java	Similar	66	53
		Dissimilar	26	14
Recoder	Java	Similar	71	84
		Dissimilar	39	46
InferCode	Java	Similar	71	81
		Dissimilar	39	31
Type4Py	Python	Similar	62	69
		Dissimilar	16	28

4.4.3 RQa$_3$: Effects of Model Capacity on Representation Quality and Downstream Task Performance

The findings from RQa$_1$ and RQa$_2$ suggest that the models studied do not fully capture the syntax of the programming language within their latent space. Instead, they tend to abstract the syntax to some degree. To explore this further, we investigate whether increasing each model's capacity, specifically by increasing the number of parameters in the layers responsible for extracting embeddings, can enhance their syntax learning capabilities. This investigation allows us to discern whether the models' limitation in representing full syntax stems from insufficient capacity or from inherent architectural constraints.

Table 4.8 presents the results of our experiments. The "Scaled Hidden Size" column indicates the number of hidden units used in the core representation learning layer(s) of each model, such as the recurrent, convolutional, or LSTM layers after scaling, as described in

Table 4.7 Result of *DeepCodeProbe*'s accuracy on recovering abstract syntactic information

Model	Task	Programming Language	Accuracy C(%)	Accuracy U(%)
AST-NN	CCD	C	99.17	62.11
		Java	97.50	61.25
SummarizationTF	Code Summarization	Java	73.64	60.97
CodeSumDRL	Code Summarization	Python	43.21	32.03
Dear	APR	Java	63.11	70.66
Recoder	APR	Java	96.40	86.33
InferCode	Code representation	Java	79.20	88.51
Type4Py	Type Inference	Python	42.50	78.83

Section 4.3.4. These layers are responsible for generating the internal embeddings used in our probing setup. Importantly, we keep the overall architecture, dataset, and training procedure fixed, modifying only the hidden dimensionality of these layers to isolate the effect of model capacity on syntax learning. We use the same $\langle d, c, u \rangle$ tuple construction and probing approach from RQa_2 to evaluate the scaled models.

Analyzing the results presented in Table 4.8, we can observe that:

- AST-NN: For both C and Java, when we scale up from the original 128 parameters to 256, we obtain an increase in the accuracy rates for d, c, and u predictions. However, as we expand the model beyond this point, the rate of improvement in accuracy diminishes. This suggests that the model's capacity to capture syntactic information does not proportionally increase with size, given the architecture in place.

- FuncGNN: Scaling the number of parameters for FuncGNN from 512 to 1024 results in no noticeable difference in the accuracy of retrieving the $\langle d, c, u \rangle$ tuples. Considering the results obtained in RQ1, where we observed that FuncGNN focuses on the edges of CFGs, by scaling up the model, we do not observe any differences in the model's capability to discern between node position and node types for clone detection. Furthermore, we observe a decrease in the model's capability in CCD, indicating overfitting, meaning that the model has reached its optimal capacity for the current architecture.

- SummarizationTF: Scaling the number of parameters from 512 to 1024, displays no improvement in increasing the accuracy of retrieving the d tuple and a decrease in the

probe's capability to predict the c and u tuples. This suggests that the model has reached its learning capacity within the current architectural constraints, and scaling it up further will only result in overfitting and diminishing returns.

- CodeSumDRL: Increasing the number of parameters from 512 to 1024 shows a noticeable improvement in the probe's accuracy in predicting the $\langle d, c, u \rangle$ tuples. Further scaling to 2048 parameters does not result in more improvement. However, no overfitting is observed. By further scaling to 4096 parameters, there's a drop in the probe's accuracy in predicting the $\langle d, c, u \rangle$tuples compared to 2048 parameters, suggesting that the model has reached its optimal capacity for the current architecture, leading to overfitting in learning syntactic representations.

- DEAR: Scaling the number of parameters from 128 to 256, as described in Section 4.3.4, results in improved accuracy across all $\langle d, c, u \rangle$ predictions. However, the rate of improvement is marginal, particularly for the d and c tuples. Specifically, we observe the highest improvement in accuracy for predicting the u tuples (6.22%), while the accuracies for d and c increase by only 4.1% and 3.99%, respectively. Further scaling beyond 256 parameters leads to overfitting, indicating that the model's capacity to effectively learn syntactic representations has reached its limit within the current architecture.

- InferCode: Scaling the number of parameters from 100 to 200 shows noticeable improvements in accuracy across all $\langle d, c, u \rangle$ tuples, especially in u tuples. Further scaling from 200 to 400 parameters continues to improve accuracy, with the accuracy for the u tuples seeing the most significant improvement. However, scaling beyond this point begins to introduce overfitting. As such, the results suggest that while InferCode benefits from initial scaling to capture syntactic information effectively, it quickly reaches its syntax learning capacity.

- Type4Py: Scaling the number of parameters from 512 to 1024 results in noticeable accuracy improvements across all $\langle d, c, u \rangle$ tuples, the u tuples' accuracy increasing to 67.9%. However, further scaling beyond this point does not yield additional gains and instead leads to overfitting. This indicates that Type4Py has reached its optimal capacity for learning syntactic and semantic representations within the current architecture.

Given the results of scaling the models and given the architecture of each model as described in Section 4.3.1, we can observe that increasing the models' size provides diminishing returns for each model after a certain point. Consistent with existing literature [11], each successive layer in a DNN learns more complex representations from the previous layers' outputs. Our

Table 4.8 Result of *DeepCodeProbe*'s accuracy on recovering syntactic information on models with higher capacity

Model	Scaled Hidden Size	Total Learnable Parameters	Programming Language	Accuracy D(%)	Accuracy C(%)	Accuracy U(%)
AST-NN	256	247,682	C	13.11	15.02	15.40
			Java	10.47	9.30	8.92
	512	434,562	C	8.62	8.73	8.73
			Java	8.94	8.82	8.95
FuncGNN	1024	485,632	Java	44.24	98.60	39.90
SummarizationTF	1024	3,553,537	Java	13.82	13.81	13.74
	2048	14,184,961		13.79	13.60	13.44
CodeSumDRL	1024	6,816,256	Python	50.41	42.75	36.72
	2048	22,545,408		50.23	42.97	36.30
Dear	256	1,395,300	Java	41.50	17.36	48.14
InferCode	200	191,500	Java	17.33	43.74	69.61
	400	422,200		18.20	50.12	71.29
Type4Py	1024	9,740,288	Python	30.01	45.52	67.90

results also show that beyond a certain size and without changing the models' architecture, their capacity to learn syntactic information stops improving and shows a decline, indicating potential overfitting. Moreover, the models' effectiveness on their original tasks shows little enhancement with increased size, suggesting that the initial parameter counts were already optimal and beneficial for maintaining the model's generalization capabilities. These observations, coupled with the results obtained from probing the original models on their syntax learning ability in the previous RQs lead us to an important conclusion: *if the task and data representations are selected adequately, there is no need to employ models that learn the full syntax of the programming language for software engineering.*

Findings 3: Scaling models trained on code provides marginal returns on their syntax learning capabilities without showing much improvement on the models' original tasks. Given that the models are trained on syntactical representations of code, and their much smaller size compared to LLMs, our results show that there is no need to have the models learn the full syntactic rules of the programming language as long as the task and data representation are selected adequately.

4.5 Discussion

Based on the experiments we conducted while replicating the results of the models under study, as described in Section 4.3.1, along with the results of our probing, we derive guidelines for software engineers to optimize data quality when training ML models on code. Our results highlight critical insights into how syntactic representations, model capacities, and probing techniques can enhance both model performance and reliability.

4.5.1 Leveraging Syntactic Representations for Efficient Model Training

In our experiments, we evaluated a range of models trained explicitly on ASTs/CFGs. While the models under study vary in their underlying architectures and downstream tasks, they commonly utilize syntactic representations rather than text tokens. Our analysis of their performance, combined with comparisons reported in their original publications [1–8], indicates that syntactically enriched representations consistently result in performance either better or comparable to token-based approaches, especially for smaller models. A key finding is that models leveraging AST/CFG representations effectively capture syntactic abstractions, enabling stronger generalization on their training objectives without the requirement of fully learning the entire programming language syntax. As such, these syntactic abstractions allow for efficient training and inference, enabling smaller models to achieve strong performance without the extensive resource demands of LLMs, which are known to be computationally intensive and prone to hallucinations [10, 242, 243]. Based on these insights, we recommend adopting AST, CFG, or syntactically derived representations when developing models intended for task-specific objectives, as they offer practical efficiency and robust performance.

Recommendation 1: We recommend that practitioners training models for task-specific software engineering objectives utilize structured syntactic representations, such as ASTs or CFGs, rather than treating code as raw text. This enables more efficient training and reduces the risk of performance degradation during deployment.

4.5.2 Task-Specific Data Representation: Adapting to Model Architectures

Our study reveals that tailoring data representations to the specific task and model architecture can significantly enhance model performance. For instance, AST-NN utilizes STs to break down complex ASTs into manageable substructures, while FuncGNN employs statement-level tokenization within CFGs to focus on control flow details. SummarizationTF and CodeSumDRL are both models trained to provide natural language descriptions of code.

This can be viewed as a translation task (by looking at code as one language and the natural language as the target language). In fact, most comment generation and code summarization approaches are conceptualized as language translation tasks in which each block of code is systematically mapped to a corresponding natural language description [10]. However, as described in Section 4.3.1 and Section 4.3.1, using AST extracted from code rather than treating code as text allows SummarizationTF and CodeSumDRL to achieve good performance while being much smaller than LLMs and still capable of learning abstractions from the programming language's syntax. Furthermore, both DEAR and Recoder operate on ASTs for APR. Similar to AST-NN, DEAR decomposes each AST into smaller subtrees, while Recoder processes the full AST structure. As discussed in Section 4.4.1, although DEAR is unable to fully capture the programming language's syntax it still achieves relatively strong performance despite having significantly fewer trainable parameters in comparison to Recoder. This further demonstrates that aligning data representations with task requirements can lead to effective and efficient model performance, especially in resource-constrained settings.

Recommendation 2: Practitioners should align their data representation strategies with the target model architecture and task requirements. For instance, AST-based representations have been shown to be particularly effective for tasks such as code summarization, comment generation, and automated program repair, where preserving syntactic structure can aid in capturing meaningful abstractions. Even when full syntactic coverage is not possible, task-aligned representations can enable smaller models to achieve competitive performance with significantly lower computational cost.

4.5.3 Enhancing Model Reliability through Probing

The need for interpretable models in software engineering is crucial, especially in safety-critical applications. Our experiments using *DeepCodeProbe* show that probing techniques can provide valuable insights into what models learn from their training data. Probing helps identify whether models are capturing syntactic abstractions or merely learning superficial patterns. Our results indicate that smaller models, although not as comprehensive as LLMs, can effectively learn meaningful abstractions from syntactic data representations. This is essential for ensuring reliability, as models with lower complexity are less prone to unpredictable behaviors, such as hallucinations, which are common in LLMs [244].

Recommendation 3: We recommend incorporating probing techniques, such as *DeepCodeProbe*, into the model evaluation pipeline to ensure that models learn meaningful abstractions rather than superficial patterns. This approach improves model reliability and helps identify potential weaknesses during deployment.

4.5.4 Scaling Model Capacity: Impact on Data Quality and Efficiency

Our investigation into the effects of scaling model capacity (addressed in RQ3) highlights a trade-off between increasing model size and maintaining efficient performance. While increasing the number of parameters can improve syntax learning, it also risks overfitting and increased inference times. Targeted scaling of specific components, such as the recurrent layers in AST-NN or graph convolution layers in FuncGNN, can yield performance gains without significantly raising computational costs.

Recommendation 4: For tasks requiring both high accuracy and resource efficiency, practitioners should selectively scale internal model components while refining syntactic representations. By focusing on layers that leverage structured data, models achieve performance gains without inflating computational costs.

4.5.5 Smaller Models vs. Large Language Models: Practical Considerations

With the rise of LLMs, there is a tendency to train/fine-tune models on extensive corpora of source code, which enables these models to implicitly learn the syntax of programming languages. This approach has shown success, as various probing studies on LLMs report that these models can represent programming language syntax in their latent space [156–158, 212]. However, LLMs are expensive to train [245], require enormous amounts of training data [208], are prone to hallucinations [246], and lack interpretability [244]. As shown by Tambon et al. [209], LLMs' lack of interpretability and hallucinations can introduce bugs or vulnerabilities in their generated code, making them unreliable for critical tasks. In contrast, the models we have studied are significantly smaller and less resource-intensive. Their training is computationally inexpensive compared to LLMs, and they can be trained on readily available data without complex pre-processing. Consequently, inference on these models is also inexpensive. They are all based on either RNNs, encoder-decoder, or Seq2Seq architectures, which are well-studied in ML and SE literature for interpretability [96]. Our study demonstrates that smaller models, trained on syntactic representations, provide a balanced approach by offering transparency, reduced computational costs, and reliable performance.

Recommendation 5: For critical software engineering tasks, practitioners should prioritize smaller, interpretable models over LLMs. This approach ensures more predictable behavior and avoids the risks associated with opaque, large-scale models.

4.6 Threats To Validity

Threats to internal validity. concern factors, internal to our study, that could have influenced our results. Our data representation scheme for each model, which is built upon AST-Probe, can be considered a potential threat to the validity of our research. Probing has been used in other contexts in NLP for uncovering what the models learn, and our results show that by extracting a syntactically valid vector representation, we can probe models for syntactical information. Additionally, our data transformation and embedding extraction for each of the models under study may not encompass all the required steps to understand the internal representations of DL models. We mitigate this threat by experimenting with multiple data representation approaches and extracting the embeddings from different layers of the models under study to have a comprehensive understanding of the probing task and the obtained results.

Threats to construct validity concern the relationship between theory and observation. The primary threat to the construct validity of our study is the probing technique we propose. This technique could potentially provide a distorted view of what models have truly learned, thereby impacting our analysis and the conclusions drawn from it. In order to mitigate this threat, we validate the extracted embeddings by comparing the cosine similarity of code clone and non-clone pairs from models trained on syntactically valid code representations. Furthermore, we establish the probe's validity by demonstrating that it does not learn new representations but reveals pre-existing information in the model's latent space. Moreover, we have conducted thorough experiments using embeddings from both trained and randomly initialized models, ensuring our probing approach robustly reflects the underlying syntactic learning of the models. Finally, our definition of 'data quality' in this study focuses on syntactic retention and abstraction capabilities. However, it does not account for semantic or context-dependent aspects of data quality, which may limit the scope of our findings.

Threats to conclusion validity concern the relationship between theory and outcome and are mainly related to our analysis done on our probes' results. To mitigate this threat, we have conducted probing on models with the same architecture but higher capacities to ensure that our conclusions and the suggested guidelines are based on both theoretical and empirical evidence.

Threats to external validity concern the generalization of our findings. We have selected models for software engineering across different tasks with different data representation schemes and trained on different programming languages, in order to have a comprehensive analysis and mitigate this threat. Furthermore, there exists the possibility that our probing approach produces different results than the ones reported on other models with relatively the same capacity but different architecture and data processing approaches. Given the breadth of existing ML models for software engineering, we selected models that reflect a diverse set of architectures, including encoder-decoder architectures, graph-based models, seq2seq models, and transformer-based models, to ensure our analysis spans a representative range of modeling strategies. Additionally, we selected models spanning five distinct software engineering tasks (code clone detection, code summarization, automated program repair, code representation learning, and type inference). This task diversity ensures that our findings are not tied to a single problem formulation but instead generalize across a wide range of software engineering objectives. Furthermore, our ability to validate and analyze model behavior required selecting models with publicly available code and data, trained on syntactically valid representations. This ensured reliable replication and that observed behaviors, especially any negative findings, were attributable to the models under study themselves rather than results of replication error. Given such limitations, the models under study cover the majority of the proposed approaches across programming languages and architectural designs. Given that our data transformation and embedding extraction are model agnostic, we believe that our probing approach can be used to probe other models as well.

Threats to reliability and validity concern the possibility of replicating this study. We have provided all the necessary details needed for replication, sharing our full replication package [230]. Additionally, since our probing approach requires re-training the models under study, our replication package includes the code, links to data repositories, and the configurations for re-training the models.

4.7 Chapter Summary

This chapter presents an empirical investigation into the quality of syntactic representations learned by ML models trained on code for software engineering tasks. We introduced *DeepCodeProbe*, a novel probing approach tailored to analyze the representations learned by smaller code models. By utilizing robust data representations (AST/CFG) and well-designed probes, we assessed the reliability, abstraction capabilities, and capacity-dependent quality of these representations. Our key findings reveal that:

- Syntactical code representations (AST/CFG) enable structured retention of syntax information, outperforming methods that treat code as plain text. By explicitly encoding structural relationships, these representations enhance ML models' ability to learn and utilize programming language syntax, offering a more reliable foundation for downstream tasks.
- Across models, learned representations capture meaningful syntactic abstractions, with robustness observed even in smaller architectures. This highlights the potential for efficient training of lightweight models without significant trade-offs in syntax abstraction quality.
- Our scaling experiments show that while increasing model capacity improves syntactic retention and consistency, it introduces diminishing returns beyond certain thresholds and increases overfitting risks. These results underscore the importance of balancing model size and training objectives to optimize data quality outcomes.

Our findings emphasize the critical role of data quality in training reliable and interpretable ML models for software engineering. Specifically:

- High-quality, syntactically valid representations (e.g., AST/CFG) significantly enhance model performance and interpretability.
- Understanding the interplay between representation quality and model architecture can guide the selection of suitable representations for specific tasks.
- The trade-offs between model capacity and representation consistency must be carefully managed to avoid overfitting while ensuring robust learning outcomes.

Additionally, alongside our study, we provide actionable guidelines for improving data quality in ML pipelines for software engineering, including recommendations for representation choice and capacity scaling. Finally, we provide a replicable framework for systematically assessing the quality of code representations, enabling practitioners to identify potential data quality issues early in the ML life cycle.

CHAPTER 5 Trained Without My Consent: Detecting Code Inclusion In Language Models Trained on Code

As detailed in Chapter 2.6.2, LLMs are increasingly integrated into the software development life-cycle. As such, ensuring compliance of LLM-generated code with licensing and copyright becomes both more important and difficult. Traditional code auditing approaches aim to verify that the produced code does not contain material from protected sources. However, the rise of LLM-based coding assistants has introduced a major challenge: LLMs are trained on massive, undisclosed datasets scraped from public codebases, making it unclear whether output code may inadvertently reproduce copyrighted material (see Chapter 3.3). Given the non-disclosure of LLMs' training datasets, traditional approaches such as code clone detection are insufficient for detecting copyright infringement.

The objective of this chapter is to introduce an approach for detecting whether a given codebase, or substantial parts of it, was included in the training dataset of an LLM without requiring access to the LLM's training dataset or weights. To address this, we present *TraWiC*, a model-agnostic and interpretable method based on membership inference for detecting code inclusion in an LLM's training dataset. *TraWiC* operates by extracting syntactic and semantic identifiers unique to each program to train a classifier for detecting likely dataset inclusion. Furthermore, *TraWiC* does not require access to the model's weights or training data, making it suitable for auditing LLMs given a black-box level of access. Our evaluation shows that *TraWiC* is capable of detecting 83.87% of codes that were used to train an LLM. In comparison, the prevalent clone detection tool NiCad is only capable of detecting 47.64%. In addition to its performance, *TraWiC* has low resource overhead in contrast to pair-wise clone detection that is conducted during the auditing process of tools like CodeWhisperer reference tracker, across thousands of code snippets.

5.1 Introduction

As described in Chapter 2.6, LLMs are being increasingly used in various SE tasks such as software development, maintenance, and deployment. However, the inaccessibility of these models' training datasets raises the important challenge of auditing the generated code, as these models are trained on code collected from publicly available sources such as GitHub, which contains licensed code (see Chapter 2.6.2). CCD techniques are one of the more prominent approaches used to safeguard against using copyrighted code during software development [247, 248]. However, to do so, having access to the codes used for training the models

(i.e., original codes) is required, which is not possible when analyzing codes generated by an LLM, as the training datasets of these models are not made publicly available.

Previous works have shown that LLMs can re-create instances from their training data [166–168]. This is known as the *memorization problem* in which ML models re-create their training dataset instead of generalizing [105]. By exploiting memorization, MIAs [106] have been shown to be effective at both extracting information from ML models and inferring the presence of specific instances in a model's training dataset (see Chapter 3.2). Inspired by these approaches, we present *TraWiC*: A model-agnostic, interpretable approach that exploits the memorization ability of LLMs trained on code to detect whether codes from a project (collection of codes) were included in a model's training dataset. To assess whether a given project was included in the training dataset of a model M, *TraWiC* parses the codes in the project to extract unique textual elements (variable names, comments, etc.) from each of them. Afterwards, these elements are masked, and the model is queried to predict what the masked element is. Finally, the generated outputs of the model M are compared with the original masked elements to determine if the code under analysis was included in the model's training dataset.

To evaluate our approach, we constructed two datasets of projects that were used to train three distinct LLMs. These datasets will act as the ground truth for inclusion detection and are comprised of over 10,700 code files from 314 projects. Our results indicate that *TraWiC* achieves an accuracy of 83.87% for dataset inclusion detection. As code clone detectors are traditionally used for code auditing [247, 249], we compare *TraWiC* against two of the widely used open-source code clone detectors, NiCad [250] and JPlag [251], to have an appropriate baseline for our evaluations. Our analysis shows that NiCad is only capable of achieving an accuracy of 47.64% for detecting dataset inclusion in an LLM's training dataset, while JPlag achieves an accuracy of 55%. Furthermore, unlike code clone detection approaches, *TraWiC* does not require pair-wise comparison between the codes to detect inclusion, which makes it more computationally efficient compared to clone detection.

To structure the work in this chapter, we define the following RQs:

RQb$_1$: *How can we detect dataset inclusion in LLMs without access to the training dataset or model weights?*

In order to answer this RQ, we introduce *TraWiC* and analyze its effectiveness in dataset inclusion detection. In order to structure this analysis, we define the following sub-RQs:

RQb$_1$-a: What is *TraWiC*'s performance on the dataset inclusion detection task?

RQb$_1$-b: How does *TraWiC* compare against traditional CCD approaches for dataset inclusion detection?

RQb$_1$-c: What is the effect of using different classification methods on *TraWiC*'s performance?

RQb$_2$: *How robust is TraWiC against data obfuscation techniques?*

In order to answer this RQ, we conduct a sensitivity analysis of *TraWiC*'s performance by testing it against various code obfuscation techniques to gauge the robustness of our framework under real-world conditions where data may be obfuscated or incomplete. Furthermore, we evaluate the importance of each feature type in the final detection outcome. To structure our analysis, we define the following sub-RQs:

RQb$_2$-a: How robust is *TraWiC* against data obfuscation techniques?

RQb$_2$-b: What is the importance of each feature in detecting dataset inclusion?

RQb$_3$: *How does TraWiC fail, and what can we learn from its failures?*

To answer this question, we investigate *TraWiC*'s failures (i.e., misclassifications) to understand when and why it produces false positives or false negatives. This includes examining both the model's behavior and the underlying properties of the code samples, providing insight into edge cases and the limits of our proposed framework.

5.2 Methodology

In this section, we first define the concepts and terminology used throughout the rest of this chapter to describe *TraWiC*, discuss the motivating example behind *TraWiC*'s design, and then show an example of how code is processed for dataset inclusion detection in our approach. We will then explain *TraWiC*'s dataset inclusion detection pipeline in detail.

5.2.1 Core Terminology and Definitions

As explained in Section 3.2, memorization can be leveraged in MIAs to investigate the presence of data records in a model's training dataset. Previous works have used *exact memorization* [252] and *name cloze* [168] approaches to analyze the outputs of a model for detecting dataset inclusion. In both works, parts of the input (i.e., token(s)) to the model are **masked**, and the model is tasked with predicting the masked tokens given the rest of the input as context. Afterwards, the model's outputs are compared to the original masked token. In these approaches, the model should not be able to predict the exact token (given that it

does not exist anywhere else in the input and is unique enough) unless it has seen the input during its training. In our work, we follow a similar rationale with modifications for code. We consider code as consisting of two distinct parts: **syntax** and **documentation**; with syntax being the code itself (which follows the programming language's structural rules) and documentation being an explanation of the syntax for future reference. From here on, we use the word *"script"* to denote a single file in a *"project"* (which can consist of multiple scripts) that contains the code alongside its documentation.

We formally define masking as follows:

Definition 1 (masking): Given input I which consists of tokens $[t_1, t_2, ..., t_n]$, and a chosen token T, masking T entails replacing all the tokens in I which match T with another token named, $MASK$. The model M is queried to predict what $MASK$ is, given I as input.

We also define exact and partial matching of masked tokens as follows:

Definition 2 (exact match): Let I be an input composed of a sequence of tokens $[t_1, t_2, ..., t_n]$ with a specific token T masked. An exact match is detected if the output of model M for the masked token T is identical to the actual token T:

$$M(I) = T$$

Definition 3 (partial match): Let I be an input composed of a sequence of tokens $[t_1, t_2, ..., t_n]$ with a specific token T masked. Given a similarity function S, a partial match is detected if the similarity score $S(M(I), T)$, between the output of the model M and the masked token T surpasses a specified threshold, L.

In alignment with established coding standards, such as those described in [253], variable/-function/class names must be selected in a manner that reflects their purpose and context within the codebase. These identifiers, beyond the common names that are universally used (e.g., using "set" for accessors or "get" for mutators), are unique to the script and its corresponding project, and reflect the developers' individual coding style. Additionally, documentation, which is an important part of the program, is closer to natural language as it is not constrained to the programming language's syntax and, as such, allows for a greater expression of the developers' individual styles [254, 255]. Therefore, in order to detect whether a project was used in a model's training dataset, we break each script in the project into 2 different identifier groups and compare the model's generations with the inputs as follows:

Syntactic identifiers are names or symbols used to represent various elements in programming languages. These identifiers are used to name variables, functions, classes, and other entities within the code. Excluding code reuse (i.e., using code written for one script in another) within similar projects, these identifiers are generally unique to their codebases. In fact, Feitelson et al. [256] have shown that there is only a 6.9% probability that two developers choose the same name for the same variable. Therefore, given the uniqueness of such identifiers, we look for exact matches between the masked tokens in the original script and the model's outputs. Specifically, we extract the following identifiers from each script:

- Variable names: Names of the variables declared and used throughout the script.
- Function names: Names of the functions declared and used throughout the script.
- Class names: Names of the classes declared and used throughout the script.

Semantic identifiers are expressions that are either used to explain the logic of code into human-readable, natural language terms (i.e., documentation) or contain natural language elements (e.g., strings). Documentation serves to clarify code for future reference or other developers, while strings are a specific data type used within the code. As Aghajani et al. [257] show, developers have different standards for "what" to document and "how" to convey the underlying information. They report that different developers apply their own terminology and style based on experience, individual style, and the codebase's context. Therefore, these identifiers are not strictly uniform between different projects developed by different developers. As such, we look for partial matches between the original script's identifiers and the model's outputs, given the individual and contextual nature of how developers use both documentation and strings.

- Strings: data structures that are used to handle textual data. For example, in Listing 3, the phrase "Hello World!" is a string.
- Statement-level documentation (e.g., comments): explanation of single or multiple statements' functionalities.
- Method/Class-level documentation (e.g., Docstrings, Javadocs): explanations of a method's or class's functionality. Some programming languages like Python or Lisp follow specific conventions and syntax for separating method/class-level documentation from statement-level documentation [258, 259], while other languages such as C or Java only establish a style guide for how they should be written and do not provide a separate syntax for them. Regardless of the programming language, established best practices require developers to thoroughly document

the inputs, operations, and outputs of a method/class [260], and they are longer and more detailed than statement-level documentation [261]. Therefore, we categorize method/class-level documentation in a different group than statement-level documentation.

From here on, we use the word *"element"* to denote a single extracted syntactic/semantic identifier. Outside of these two identifier groups and their constituent elements, what remains in the code is either related to the programming language's syntax or operations that execute the logic of the code.

Finally, we define the prefix and suffix of an element as follows:

Definition 4 (prefix and suffix): Let S be a script composed of a sequence of tokens $[t_1, t_2, ..., t_n]$ with a specific token T masked. We consider all the tokens in S that come before T from t_1 as prefix and all the tokens that immediately come after T until t_n as suffix.

To determine whether a script was included in the training data of a given model, we apply the *Fill-In-the-Middle* (FIM) technique, which is commonly used in both pre-training and fine-tuning of LLMs [262]. This technique consists of breaking a script into three parts: prefix, masked element, and suffix. The model is then tasked to predict the masked element given the prefix and suffix as input. In the next section, we present an example of how an incoming script is processed for detecting dataset inclusion.

5.2.2 Motivating Example

In this section, we present how *TraWiC* differs from and improves the previous MIA approaches for code. Consider model M to be a model trained on code with project P that contains the script presented in Listing 3 (C) being included in M's training dataset. Detecting both P and C's inclusion in M's training dataset using the previous approaches (exact memorization [252], name cloze [168]) would require the following steps:

- If the task is next token prediction using exact memorization or other similar approaches, then all scripts in P would be broken down into multiple, separate parts. For each generated part, the model would be tasked with predicting only the next token.
- If the task is name cloze detection or other similar approaches, each script in P would be broken down into multiple prompts, and for each prompt, the model will be tasked with predicting the masked token.

Both approaches introduce multiple points of failure for inclusion detection and require a large number of costly inference calls on the LLM. We further describe the details in the rest of this section.

```python
def print_input_string(input_string):
    """
    This function prints the input string.
    Args:
        input_string(str): the string to be printed
    """
    # store the input string in a variable
    dummy_variable  = input_string
    # print the input string
    print(input_string)

def add_variables(a, b):
    """
    This function adds two variables.
    Args:
        a (float): first variable
        b (float): second variable
    Returns:
        float: result of the addition
    """
    # add the two variables and
    # store the result in a new variable
    c = a + b
    # return the result
    return c

print_input_string("hello World!")

result = add_variables(a=1, b=2)
```

Listing 3: An example of a Python script.

Next token prediction

As an example, let us consider the exact memorization approach. Here, we would be required to break down each script in P into n different parts, with n being the number of tokens in the script, and call the model n times in order to get the model's predictions for each token. After collecting the outputs of M for each input, one needs to check for exact matches for each of the predicted tokens. While this approach might be useful for models that are only trained on non-code textual inputs (e.g., blogs, books, forums, etc.), this would not be useful

on code, especially on semantic elements, given how developers follow different styles for writing them [257]. Furthermore, given the number of parameters of the LLM under study, inference on M can become extremely costly and time-consuming. Following the running example in Listing 3, consider the docstring for the function `add_variables`. If M generates outputs that differ from the target tokens, the exact memorization approach would return a negative response. However, M has predicted a token for a semantic identifier which is closer to natural language, can be worded in similar ways without breaking the code, and is close to what was included in the original script. Furthermore, employing next token prediction results in losing all the context that comes after the token that is to be predicted by M. Therefore, for tokens that are at the beginning and near the middle of the code, the input to the model will not contain the entirety of the code, and therefore M will be prone to generating tokens that are not similar to what is present in the training data and result in false negatives.

Name cloze prediction

Using name cloze prediction, similar to exact memorization, would entail breaking down the code into n parts, constructing n prompts, and calling the model on each prompt. While this approach can be useful for detecting rare names in the model and provides context about what comes after the target token prediction, it is based on an instruct prompt and, therefore, sensitive to how the prompt is designed. As shown by Srivastava et al. [146], LLMs are sensitive to changes in prompts. Such sensitivity may cause the model to output false negatives or positives. Furthermore, similar to the exact memorization approach, name cloze approaches check for exact matches between what the model predicts and the target token and therefore face the same challenge of resulting in false negatives when M's outputs differ slightly in wording compared to what was originally observed during training.

In comparison, *TraWiC* benefits from the best aspects of both approaches. Specifically, *TraWiC* is not dependent on the prompt, similar to exact matching approaches, while providing the context of the entire code to the model, similar to name cloze approaches. Furthermore, *TraWiC* also introduces a new method for ensuring the minimization of false negatives as much as possible during the checking process and performs fewer calls on the LLM, making it more efficient. Following the running example presented in Listing 3, *TraWiC* requires only 13 calls on LLMs in contrast to 82 calls of the previous approaches (one call for each token). Specifically, by looking for exact similarities for parts of the input that are required to be exactly the same (syntactic elements) and fuzzy matching for parts of the input that can differ (semantic elements), we check the model's outputs for code inclusion on 6 differ-

ent levels. Furthermore, previous approaches rely solely on the number of matches without employing any comparison of the obtained results with previous observations. Therefore, by using a separate classifier trained on previous observations, we add another level of accuracy compared to exact matching models' outputs by identifying patterns and variations that may not be captured through exact matching alone. As a result, *TraWiC* is much more efficient, accurate, and suited for detecting code inclusion in a model's training dataset compared to the previously proposed approaches.

5.2.3 End-to-End Data Processing Example

In this section, we present an example of how *TraWiC* processes the scripts in a project for dataset inclusion detection.

```
{
  "Variable names": ["dummy_variable", "a", "b", "c", "result"],
  "Function names": ["print_input_string", "add_variables"],
  "Class names": [],
  "Strings": ["'Hello World!'"],
  "Statement-level documentation":
      [
        "store the input string in a variable", "print the input string",
        "add the two variables and store the result in a new variable",
        "return the result"
      ],
  "Method/Class-level documentation":
      [
        "This function prints the input string.\n\nArgs:\n    input_string ...",
        "This function adds two variables.\n\nArgs:\n    a (float): ..."
      ]
}
```

Listing 4: JSON representation of extracted identifiers from the script in Listing 3.

Listing 3 shows an example of a Python script. This script consists of 2 functions, 3 variable declarations, and corresponding docstrings and comments inside each function. Listing 4 displays all the syntactic and semantic identifiers extracted from the script in Listing 3. Following the example in Listing 3, Listings 5 and 6 show the constructed prefix and suffix for the variable `dummy_variable`, respectively. Note that we keep all the information from the original script and only mask the element itself. This process is repeated for every semantic and syntactic element extracted from the input script.

By breaking each script into multiple (prefix, suffix) pairs, we aim to leverage the memoriza-

```
def print_input_string(input_string):
    """
    This function prints the input string.
    Args:
        input_string(str): the string to be printed
    """
    # store the input string in a variable
```

Listing 5: Prefix generated for variable `dummy_variable` from Listing 3.

tion capability of LLMs. More specifically, as reported by Carlini et al. [167], as the capacity (i.e., number of parameters) of a language model increases, so does its ability to memorize its training dataset instead of generalizing. Therefore, by using a large part of the script that was present in the model's training dataset as input, and only masking a small piece (i.e., a single element) to be used with the FIM technique as defined in Section 5.2.1, the model would likely be pushed in the direction of generating an output that is the same or highly similar to the original masked token if the script was included in its training dataset [252].

```
    = input_string
    # print the input string
    print(input_string)

def add_variables(a, b):
    """
    This function adds two variables.
    Args:
        a (float): first variable
        b (float): second variable
    Returns:
        float: result of the addition
    """
    # add the two variables and store
    # the result in a new variable
    c = a + b
    # return the result
    return c

print_input_string("hello World!")

result = add_variables(a=1, b=2)
```

Listing 6: Suffix generated for variable `dummy_variable` from Listing 3.

Once we have collected all the model's outputs for the extracted elements of a script, we look

for the degree of similarity between the generated outputs and the original masked elements to determine dataset inclusion which we will explain in detail in Section 5.2.4. After obtaining the results of the similarity comparison, a classification model predicts whether a project was included in a model's training dataset, given the results of the similarity comparison as input.

5.2.4 Dataset Inclusion Detection

In this section, we will explain how *TraWiC* can be used for the task of dataset inclusion detection in detail. *TraWiC*'s pipeline consists of four stages: pre-processing, inference, comparison, and classification. Figure 5.1 provides an overview of our approach.

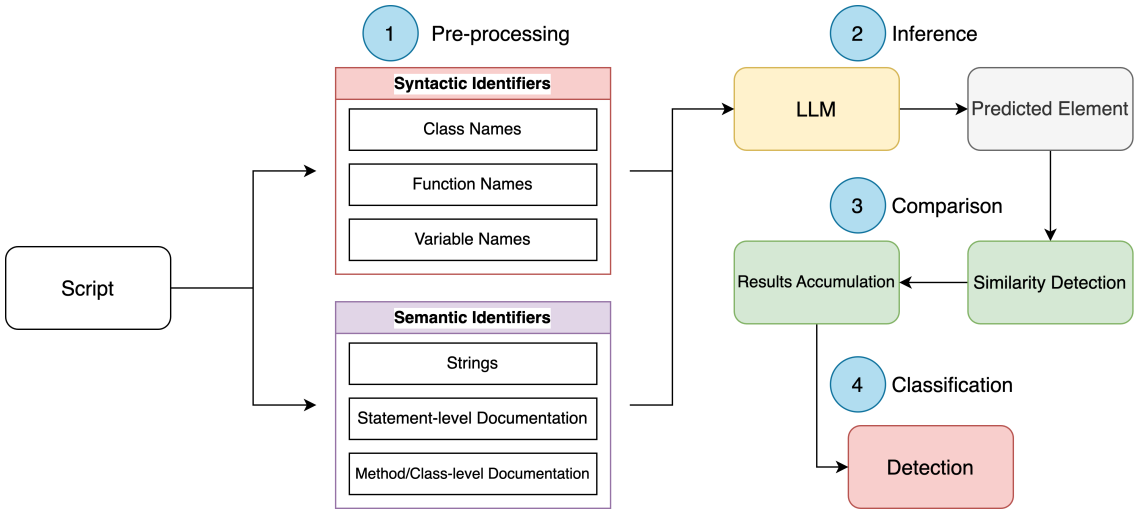


Figure 5.1 *TraWiC*'s dataset inclusion detection pipeline

Pre-processing

As stated in Section 5.2.1, we break each script into its six constituent elements (i.e., syntactic/semantic identifiers). From each script, we extract the elements, and for each extracted element, we break down the script into two parts: prefix and suffix. By creating a prefix and suffix pair for each extracted element, we will be able to query the model to predict what comes in between the prefix and suffix. Algorithm 8 describes this pre-processing step in detail.

Inference

After breaking the script into (prefix, suffix, masked element) tuples as shown in Algorithm 8, the given model is queried to predict the masked element given prefix and suffix as input.

Algorithm 8 Pre-processing

Require: SCRIPT ▷ An entire script from a project. A project can contain many scripts.
Ensure: ProcessedScript ▷ A list containing all the (prefix, suffix) tuples generated for each element from the input script.

```
1: procedure ProcessScript(SCRIPT)
2:     identifiers ← ExtractIdentifiers(SCRIPT)  ▷ Extract all syntactic and semantic identifiers from SCRIPT
3:     ProcessedScript ← Empty list
4:     for all i ∈ identifiers do
5:         (p_i, s_i) ← GeneratePrefixAndSuffix(i)
6:         Append (p_i, s_i, i) to ProcessedScript
7:     end for
8:     return ProcessedScript
9: end procedure
```

From this tuple, the prefix and suffix are used for inference on the model, while the masked element will be used in the next step for comparing the model's outputs to the original masked element. This process is repeated for all of the tuples extracted from a script. Algorithm 9 describes this process. Once we have collected all the model's predictions for each masked element, we move to the next step in which we compare the model's predictions with the original elements.

Algorithm 9 Inference

Require: ProcessedScript ▷ Output of Algorithm 8.
Ensure: ModelPredictions ▷ A list containing all of the given model's predictions for the masked element.

```
1: procedure PredictMaskedElements(ProcessedScript)
2:     ModelPredictions ← Empty list
3:     for all (p_i, s_i, i) ∈ ProcessedScript do
4:         o_i ← Predict(p_i, s_i)
5:         Append (o_i, i, type_i) to ModelPredictions
6:     end for
7:     return ModelPredictions
8: end procedure
```

Comparison

Given the outputs of the model for each element, we define two different comparison criteria, given an element's type:

Exact match (see Definition 2) For syntactic identifiers (i.e., Variable, Function, Class names), we look for an **exact** match. If the model generates the exact same token(s),

we count it as a *"hit"*, which is a binary flag that indicates whether the generated output matches the desired output.

Partial match (see Definition 3) For semantic identifiers (i.e., strings, documentation), to assess the similarity between two sequences (series of tokens), we use the Levenshtein distance (edit distance) score [263]. This metric measures the number of single-character edits that are required to change one sequence to the other. These elements are closer to natural language and do not require the syntactic restrictions required for code (incorrect/inaccurate generations will not result in syntax errors). Therefore, we compare the generated output with the original elements and calculate the edit distance between the two. Here, we define a threshold, and edit distances higher than this threshold will be considered as a *hit*. The final threshold was chosen empirically to provide a balance between semantic similarity and performance gains. We present the results with different thresholds in Section 5.4.

The other metrics that can be used for this objective are based on semantic similarity, where a semantic score is calculated between the vector representations of two strings (e.g., cosine similarity, TF-IDF distance, etc.) [264]. We choose the edit distance metric as we do not aim to inspect whether the model's output is similar to the original masked element in meaning, but in syntactic structure. Therefore, analyzing the semantic distance would not be helpful for this task.

After processing the model's outputs as described above, we normalize the hit numbers by dividing the number of hits by the total number of checks for each type of identifier. Algorithm 10 shows the comparison process in detail.

Table 5.1 shows a sample of the dataset that is constructed for dataset inclusion detection as an output of the comparison process for the example presented in Section 5.2.2.

Table 5.1 Data representation sample for the example presented in Section 5.2.2

Script Name	Class Hits	Function Hits	Variable Hits	String Hits	Comment Hits	Docstring Hits
sample_project/sample.py	0	1.0	0.08	0.4	0.66	0.0

Classification

With our extracted features, the classifier's task is to detect whether a script was included in the LLM's training dataset, which makes this a binary classification problem. Any classification method, such as decision trees or DNNs, can be used based on the desired performance. We experiment with different classification methods as explained in Section 5.4.1

Algorithm 10 Comparison

Require: ModelPredictions ▷ Output of Algorithm 9.
Ensure: Hits ▷ Dictionary of hit counts for each identifier type.

```
function UpdateHits(Hits, key, result)
    Hits[key] ← Hits[key] + result
    return
end function
procedure EvaluatePredictions(ModelPredictions)
    Hits ← {VariableHits: 0, FunctionHits: 0, ClassHits: 0, StringHits: 0, CommentHits: 0, DocStringHits: 0}
    for all (o_i, i, type_i) ∈ ModelPredictions do
        if type_i is variable/function/class name then
            if o_i = i then
                result ← 1
            else
                result ← 0
            end if
        else if type_i is string/comment/docstring then
            if SemanticComparison(o_i, i) ≥ SemanticThreshold then
                result ← 1
            else
                result ← 0
            end if
        end if
        UpdateHits(Hits, type_i, result)
    end for
    return Hits
end procedure
```

Algorithm 11 Classification

Require: Hits ▷ Output of Algorithm 10.
Ensure: ClassificationResults ▷ A binary variable: 1 indicates inclusion in the dataset, 0 otherwise.

```
procedure PredictDatasetInclusion(Hits)
    ClassificationResults ← ClassificationModel(hitType, hitCount)
    return ClassificationResults
end procedure
```

5.3 Experimental Design

In this section, we explain our experimental design, including the construction of the dataset used as the ground truth for validating our approach, the LLM under study, the classifier used for detecting dataset inclusion, and details of *TraWiC*'s performance comparison with

a state-of-the-art CCD approach. To that end, we remind our RQs and their objectives:

RQb$_1$: *How can we detect dataset inclusion in LLMs without access to the training dataset or model weights?*

In order to answer this RQ, we introduce *TraWiC* and analyze its effectiveness in dataset inclusion detection. In order to structure this analysis, we define the following sub-RQs:

RQb$_1$-a: What is *TraWiC*'s performance on the dataset inclusion detection task?

RQb$_1$-b: How does *TraWiC* compare against traditional CCD approaches for dataset inclusion detection?

RQb$_1$-c: What is the effect of using different classification methods on *TraWiC*'s performance?

RQb$_2$: *How robust is TraWiC against data obfuscation techniques?*

In order to answer this RQ, we conduct a sensitivity analysis of *TraWiC*'s performance by testing it against various code obfuscation techniques to gauge the robustness of our framework under real-world conditions where data may be obfuscated or incomplete. Furthermore, we evaluate the importance of each feature type in the final detection outcome. To structure our analysis, we define the following sub-RQs:

RQb$_2$-a: How robust is *TraWiC* against data obfuscation techniques?

RQb$_2$-b: What is the importance of each feature in detecting dataset inclusion?

RQb$_3$: *How does TraWiC fail, and what can we learn from its failures?*

To answer this question, we investigate *TraWiC*'s failures (i.e., misclassifications) to understand when and why it produces false positives or false negatives. This includes examining both the model's behavior and the underlying properties of the code samples, providing insight into edge cases and the limits of our proposed framework.

5.3.1 Models Under Study

For evaluating our approach, without loss of generality, we have selected three distinct LLMs. Namely, SantaCoder [125], Mistral 7B [265], and LLaMA-2 [130]. We expand more on each model in the following.

SantaCoder

SantaCoder is an LLM trained by HuggingFace on TheStack for program synthesis and supports Python, Java, and JavaScript programming languages. This model was chosen as one of the LLMs under study for the following reasons:

- TheStack is publicly available. Therefore, we can inspect the data and confirm the presence of a script in SantaCoder's training dataset.
- SantaCoder's data cleaning procedure is clearly explained and replicable with different cleaning criteria being applied [125]. To clean the data for SantaCoder's training dataset, its developers have used: 1) "GitHub stars" which filters repositories based on the number of their stars, 2) "Comment-to-code-ratio" which considers the script's inclusion based on the ratio of comment characters to code characters, and 3) "more near-deduplication" which employs multiple different deduplication settings. Hence, given that we have access to the original dataset, we can reproduce the dataset that the model was trained on by following the same procedures.
- SantaCoder is trained for code completion, using the FIM objective. As such, it natively supports the FIM task that is used for *TraWiC*.

HuggingFace has released multiple versions of SantaCoder. Each model is trained on data that was constructed using different data-cleaning criteria. For our experiments, we chose the `comment_to_code` model[1] because the comment-to-code ratio for this model is calculated by "using the `ast` and `tokenize` modules to extract docstrings and comments from Python files " [125]; which allows us to accurately reproduce the data. Following the data cleaning steps outlined in [125], the training dataset for this version was constructed by filtering any script in TheStack that has a comment-to-code ratio between 1% and 80% on a character level after the standard data cleaning steps (removing files from opt-out requests, near-deduplication, PII-reduction, etc).

LLaMA-2

LLaMA-2 is a family of LLMs developed by Meta [130]. LLaMA-2 was pre-trained on 2 trillion tokens of data, has a context length of 4096 tokens, and uses Grouped-Query Attention (GQA) [266]. This family of LLMs comes in different sizes (7B, 13B, 34B, and 70B), and both the pre-trained and instruct models have been released on HuggingFace[2]. The pre-trained

[1] https://huggingface.co/bigcode/santacoder

[2] https://huggingface.co/collections/meta-llama/llama-2-family-661da1f90a9d678b6f55773b

base versions provide a foundation for further fine-tuning according to specific downstream tasks while the fine-tuned versions allow for enhanced performance in conversational applications. For our experiments, we use the pre-trained version with 7 billion parameters and fine-tune it on a dataset that we have constructed.

Mistral 7B

Mistral 7B [265] is a 7-billion-parameter LLM developed by the Mistral research group, trained for language modeling tasks such as chat, code completion, and complex reasoning. This model uses GQA [266] similar to LLaMA-2 and additionally uses Sliding Window Attention (SWA) [267], which allows for faster inference speed and reduced computational costs. Alongside its fast inference, Mistral also outperforms LLaMA-2 13B on all evaluation benchmarks [265]. Similar to LLaMA-2, Mistral 7B has been released in two versions: A pre-trained base and a fine-tuned version. For our experiments, we chose the pre-trained version[3] for its flexibility and potential for fine-tuning.

Both Mistral 7B and LLaMA-2 were selected for this study due to their widespread use as foundational models for fine-tuning [268]. Unlike models already optimized for specific downstream tasks, these pre-trained base models allowed us to demonstrate the effectiveness of our approach on versatile, general-purpose language models. Their selection also provided a balanced representation of different model architectures and training processes [268]. Even though the developers of both LLaMA-2 and Mistral provide some general descriptions about the datasets that were used to train the models, unlike SantaCoder, the datasets that were used to train these models are not publicly available. Moreover, unlike SantaCoder, neither Mistral nor LLaMA-2 supports the FIM objective natively. Therefore, for our experiments, we first construct a dataset and fine-tune both models on the constructed dataset. We explain our dataset construction in Section 5.3.2 and our fine-tuning process in Section 5.3.3.

5.3.2 Dataset Construction

In order to assess our approach's correctness, we need to have access to the dataset that an LLM was trained on. By knowing which scripts/projects were used for training the model, we can construct a ground truth for training and validating the classifier for dataset inclusion detection. To do so, we use two datasets as follows.

[3]https://huggingface.co/mistralai/Mistral-7B-v0.1

TheStack

TheStack[4] is a large dataset (3 terabytes) of code collected from GitHub by HuggingFace[5]. This dataset contains codes for 30 different programming languages and 137.36 million repositories [134]. We focus on the Python repositories in this dataset as the data cleaning process used for SantaCoder, which we outlined in Section 5.3.1, is accurately replicable for Python. Given the large number of Python scripts and the data cleaning approach used for producing the training data for the LLM, some scripts within a project might be included in the LLM's dataset, while others might be excluded. Therefore, we first filter the Python projects by considering a minimum of 10 and a maximum of 50 scripts per project. We focus on projects of such scale for the following reasons:

- Our aim is to increase the likelihood that some scripts within a project might be included in the training dataset of the LLM, while some from the same project might not. This will allow us to have a more diverse dataset for both script-level and project-level dataset inclusion detection as explained in Section 5.3.5.
- As our primary objective is to detect a project's inclusion in the training dataset of an LLM, by not including overly large projects that contain many scripts, we will have a larger number of projects in our dataset (e.g., instead of having a project with 200 scripts, we can have 5 projects with 40 scripts).
- As we outline in Section 5.3.1, we have chosen a data cleaning criterion based on documentation to code ratio. Therefore, to construct a balanced dataset where projects are not documented enough or too documented, we do not include overly small projects in our dataset. As reported by Mamun et al. [269], there exists a correlation between the size of a software project and the amount of documentation written for it. Therefore, by not including overly small projects, we will increase the likelihood of having high-quality, well-documented scripts in our dataset.

It should be noted that SantaCoder was trained on all of the Python, Java, and JavaScript codes in TheStack after applying the data cleaning criterion. We then randomly sample from the remaining projects for constructing the dataset for inclusion detection for *TraWiC*, NiCad, and JPlag (as explained in Sections 5.2.4 and 5.3.6).

[4]https://huggingface.co/datasets/bigcode/the-stack
[5]https://huggingface.co/huggingface

GitHub repositories

As mentioned in Section 5.3.1, unlike SantaCoder, the datasets that were used to train Mistral and LLaMA-2 are not publicly available, and the developers of both models do not explicitly declare what data was used to train their models. As such, in order to evaluate our approach on much larger and more capable models, we construct a dataset from GitHub repositories that were published **after** the release of both Mistral and LLaMA-2 (September 2023). Similar to the dataset used for SantaCoder, in order to have a fixed approach and compare the models methodologically, we focus on repositories that contain Python code. As these repositories were published after these models' release, we can be certain that they were not used as the training data for the models under study. After collecting these repositories, we fine-tune the models on the dataset. As such, we will have a ground truth for both Mistral and LLaMA-2, which contains codes that the models have seen during their training and codes that were not included in their training dataset. The resulting dataset contains over 1355 Python scripts from 51 repositories, out of which 20000 data points (which we will further explain in Section 5.3.3) were constructed. 10000 data points were used for fine-tuning the models, and the remaining 10000 were used for evaluation of *TraWiC* itself (therefore not included in the model's fine-tuning dataset). The list of repositories that were used to fine-tune the models is available in our replication package [270].

5.3.3 Fine-tuning

As described in Section 5.3.1, in contrast to SantaCoder, Mistral, and LLaMA-2 are not trained to support the FIM objective natively. So, we use the dataset that we constructed from GitHub repositories as described in Section 5.3.2, to fine-tune the models for the FIM objective. Doing so has two benefits:

- Constructing our own dataset allows for having a ground truth for determining whether a script/project was included in a model's training dataset. Therefore, we can evaluate *TraWiC* on more capable models without the loss of generality.
- As the pre-trained versions of Mistral and LLaMA-2 do not support the FIM objective, using our own constructed dataset for fine-tuning the models allows these models to support the FIM task.

In order to fine-tune the models for the FIM objective, we use the same prompt format as the one that was used to train SantaCoder. Listing 7 displays the prompt that was used to

fine-tune the models with `prefix` and `suffix` being constructed as explained in Section 5.2.1 and `infill` being the infilling objective which the model is tasked to predict.

```
<fim-prefix>{prefix}<fim-suffix>{suffix}<fim-middle>{infill}<endoftext>
```

Listing 7: The prompt format used for fine-tuning the models under study.

As an example, consider the script that was presented in Listing 3. Listing 8 presents a single data point extracted from this script after extracting the syntactic and semantic identifiers from this script as displayed in Listing 4.

```
"prefix": "def print_input_string(input_string): \"\"\" ...\"\"\""
"infill": "dummy_variable"
"suffix": " = input_string \n # print the input ...."
```

Listing 8: The data point format used for fine-tuning the models under study.

Following this approach, we go over the collected dataset and generate the fine-tuning dataset. The models are only trained on Python scripts, and if codes from other languages exist in the dataset, they are not included in the fine-tuning dataset. In order to fine-tune the models on our dataset for the FIM objective, we use the Quantized Low-Rank Adaptation (QLoRA) [271] approach, which is a parameter-efficient fine-tuning (PEFT) method [272]. Furthermore, in order to have comparable results for both Mistral and LLaMA, both models were fine-tuned on the same dataset. The codes and the data that were used for training these models are available in our replication package [270], allowing for replication of our fine-tuned models and *TraWiC*'s results.

5.3.4 Classifier for Dataset Inclusion Detection

We have chosen to use random forests as the classifier for our experiments for the following reasons:

- We emphasize the need for our approach to maintain a high degree of interpretability. By using multiple decision trees (which are inherently interpretable) that comprise the random forest, this type of classifier provides interpretability alongside high classification performance.
- Ensemble methods, such as random forests, are known for their superior performance over single predictors due to their ability to reduce overfitting and improve generaliza-

tion. They are also more computationally efficient compared to the intensive resource demands of training and inference of DNN models [11].

To ensure a thorough comparative analysis of classifiers, we have also experimented with XGBoost (XGB) and Support Vector Machine (SVM) classifiers. For optimal classifier selection, we employed the grid search methodology, which involves an exhaustive exploration of hyperparameters [11], where for each model, we evaluated various combinations of settings as described below. The best models were selected based on the achieved F-score on the test dataset. All scripts for training these models are available at [270].

Random Forest

- Number of estimators: This hyperparameter is used to indicate the number of trees in the forest. We used values of 50, 100, and 200.
- Number of features for best split: The number of features to consider when looking for the best split. We used the values "sqrt", indicating the square root of the number of features in the input; and "log2", indicating the binary logarithm of the number of features in the input.
- Max tree depth: Maximum depth allowed for a single tree in the forest. We used the values of 10, 20, and 30.
- Split criterion: The function to measure the quality of a split. With "gini" indicating Gini impurity and "entropy" indicating Shannon information gain.

SVM

- C: The regularization parameter with the strength of regularization being inversely proportional to C. We used values of 0.1, 1, 10, and 100.
- Kernel: Specifies the kernel type to be used in the algorithm. We used radial basis and linear functions.
- Kernel Coefficient: The kernel coefficient is a parameter that determines the shape of the kernel function used to map the input data into a higher-dimensional feature space. We used values of 1, 0.1, 0.01, and 0.001.

XGB

- Learning rate: Also denoted as "eta", is the step size shrinkage used in the update. We used values of 0.01, 0.1, and 0.5.
- Max depth: maximum depth of a tree. We used values of 3, 5, and 7.
- Number of estimators: Number of boosting rounds or trees to build. We used values of 50, 100, and 200.
- Subsample: Subsample ratio of the training instances. For example, a subsample of 0.5 means that we randomly sample half of the training data before growing the trees. We used values of 0.6, 0.8, and 1.0.
- Subsample ratio of columns: Subsample ratio of columns when constructing each tree. Subsampling occurs once for every tree constructed. We used values of 0.6, 0.8, and 1.0.

5.3.5 Detecting Dataset Inclusion

Depending on the data cleaning criteria used for filtering the scripts for training the LLM as mentioned in Section 5.3.1, some scripts in a project may not be included in the model's training dataset, e.g., out of 20 scripts in a project, some may not pass the data filtering criteria. Therefore, we outline two different levels of granularity for predicting whether a project was present in a model's training dataset:

- File-level granularity: We determine whether a single script has been in the LLM's training dataset.
- Repository-level granularity: We analyze all the scripts from a repository and consider that the repository has been included in the LLM's training dataset if the classifier predicts a certain amount of scripts out of the total number of scripts in the repository as being in the training dataset.

To analyze the impact of edit distance on the classifier's performance, we define multiple levels of edit distance for a thorough sensitivity analysis. Our analysis for SantaCoder was conducted on 263 repositories extracted from TheStack, totaling 9,409 scripts, with 1,866 scripts not included in SantaCoder's training dataset. For both LLaMA-2 and Mistral, we used the same dataset that we fine-tuned the models on, designating half as the ground truth for scripts included in the training dataset, and the other half as scripts not included.

This means the classifier was trained on 1,355 scripts from 51 repositories. Out of which, 25 repositories and their corresponding scripts were used for fine-tuning the models (thus seen by the model during training), while 26 repositories and their corresponding scripts were used as negative instances (not seen by the model during training).

5.3.6 Comparison With Other Clone Detection Approaches

In software auditing, one of the approaches for detecting copyright infringement is using CCDs to check the code against a repository of copyrighted code [247]. CCD is an active area of research, and different approaches are proposed for the detection of clones, especially those of *Type 3* and *Type 4* [218]. This process is carried out through a pair-wise comparison of the codes in a project against another dataset that contains copyrighted codes. As detailed in Chapter 2.3.2, many different approaches are proposed and available for CCD [218]. Out of the proposed approaches, we choose NiCad [250] and JPlag [273] for comparison with *TraWiC* on the dataset inclusion task, for the following reasons:

- **Python support:** Out of the proposed, available approaches, NiCad and JPlag are the only ones that support clone detection for Python scripts.
- **Availability:** A few of the proposed approaches are available for use. Furthermore, most of them require expensive re-training on new datasets. However, both NiCad and JPlag are open-source and do not require re-training (as opposed to the proposed ML-based solutions).
- **Syntactic errors in generated codes:** Most of the proposed approaches for CCD based on DNNs, extract information such as ASTs from the code, which can only be done if the generated code is syntactically valid. Even though LLMs have shown a high capability in generating correct code, limiting CCD to codes that are syntactically valid would severely reduce the size of the dataset that can be used for this study.

NiCad employs a parser-based, language-specific approach using the TXL transformation system to extract and normalize potential code clones. It is capable of detecting code clones up to Type 3 [250]. JPlag [273] compares the similarity of programs in two phases: converting programs into token strings and comparing these strings using the "Greedy String Tiling" [251] algorithm. This algorithm identifies the longest common substrings between two token strings and marks them as tiles. The similarity value is determined by the percentage of tokens covered by these tiles. JPlag's tokenization is language-dependent, and it prioritizes tokens that reflect a program's structure while ignoring superficial aspects like whitespace

and comments. For the purpose of this study, we use the latest version of JPlag[6]. As NiCad and JPlag are designed to detect clones between scripts, they operate differently compared to our approach, where we inspect the similarity between one or multiple generated tokens. Therefore, to determine whether a code was in a model's training dataset with NiCad/JPlag, we query the model to generate code snippets and compare the generated codes against each snippet that we know exists in the model's training dataset. Algorithm 12 shows the steps for dataset inclusion detection using NiCad and JPlag. We generate code snippets instead of tokens since comparing scripts to each other using NiCad and JPalg will result in positive clone matches as a large portion of both scripts will be the same, as shown in Figure 5.2.

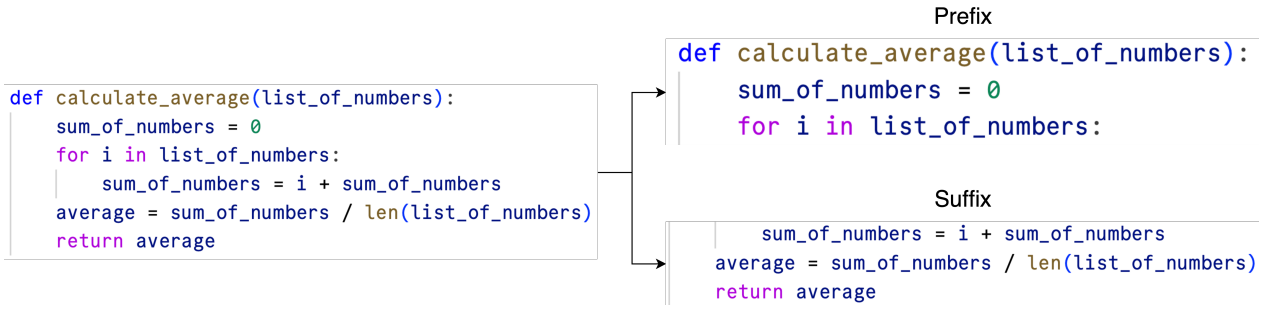


Figure 5.2 An example of how prefix and suffix are generated for NiCad and JPlag

Our detailed process is as follows:

Algorithm 12 Dataset Inclusion Detection Using NiCad/JPlag

Require: SCRIPT ▷ SCRIPT is an entire script from a project. A project can contain many scripts.
Ensure: ClassificationResults ▷ A binary variable: 1 indicates inclusion in the dataset, 0 otherwise.

```
procedure DetectInclusion(SCRIPT)
    (prefix, suffix) ← ExtractPrefixAndSuffix(SCRIPT)
    GeneratedSuffix ← Predict(prefix)
    ClassificationResults ← Empty list
    for all s ∈ dataset do
        r ← NiCad(s, GeneratedSuffix)
        Append r to ClassificationResults
    end for
    return ClassificationResults
end procedure
```

[6]https://github.com/jplag/JPlag/releases/tag/v5.0.0

Pre-processing

We extract the functions and classes that are defined in a script by parsing its code. For example, the script in Listing 3 has 2 functions. After doing so, we break each function/class into two parts based on its number of lines. Figure 5.2 displays an example of how this process is carried out. This process is repeated for every function and class that is extracted from the script. We consider the first half as the prefix and the second half as the suffix.

Inference

After generating prefix and suffix pairs, we query the model with the prefix and have the model generate the suffix. Note that here, we do not restrict the model to predict the masked element by giving it the suffix. Instead, we allow the model to generate what comes after the prefix until the model itself generates an "end-of-sequence" token.

Comparison

TheStack dataset contains 106.91 million Python scripts [134]. Therefore, comparing each generated script by the model under study in a pair-wise manner against all the other scripts present in the training dataset is computationally infeasible. As such, we compare each generated script with those of multiple randomly sampled repositories from the model's training dataset to detect code clones across projects. Here, we aim to investigate whether the model generates codes from its training dataset when it is given a large part of a script as input. If NiCad/JPlag detects a clone between the generated script and another script from the sampled projects, we consider it as a *hit* for detecting that the project was included in the model's training dataset. As we are sampling randomly, there exists the possibility that the model generates code from its training dataset, but the generated code wasn't compared against the project for which it was generated. Therefore, to decrease the likelihood of such cases, we do the following:

- We follow the same project selection criteria as described in Section 5.3.2 for both selecting the scripts to generate code from and comparing the generated codes against. By only sampling projects that contain a minimum of 10 and a maximum of 50 scripts, we aim to decrease the amount of missing *hits*.
- As we expand upon in Section 5.4.1, we experiment with multiple random sampling ratios in order to have a comprehensive sensitivity analysis.

5.4 Results and Analysis

In this section, we present our experimental results. We begin by laying out our experiment details, continue with presenting the results of our approach compared against a baseline, and conclude this section by conducting an error, feature importance, and sensitivity analysis. Our replication package is publicly accessible online [270].

5.4.1 RQb$_1$: Detecting dataset inclusion in LLMs without access to the training dataset or model weights

The results detailed in Tables 5.2, 5.3, and 5.4 showcase the performance of *TraWiC*'s dataset inclusion detection when applied with varying edit distance thresholds. The "Considering only a single positive" criterion showcases the results on file-level granularity. We consider multiple inclusion criteria for predicting whether a repository (i.e., project) was included in the model's training dataset as outlined in "Considering a threshold of 0.4/0.6". "Precision" measures the percentage of correctly predicted positive observations out of all predicted positives, reflecting the classifier's accuracy in making positive predictions. "Accuracy" gives an overall assessment, showing the ratio of correctly predicted observations to the total predictions made. The "F-score" balances precision and sensitivity (recall), offering a comprehensive metric, especially for uneven datasets. "Sensitivity" assesses the model's capability to correctly distinguish actual positive cases, while "Specificity" measures its performance in distinguishing true negatives. We focus on the F-score as the main performance metric. After F-score, we focus on sensitivity as our goal is to detect dataset inclusion; therefore, the higher the classifier's ability to detect true positives, the more suited to our purpose it is.

RQb$_1$-a: What is *TraWiC*'s performance on the dataset inclusion detection task?

We observe that the performance of our models varies based on the repository inclusion criterion and the edit distance threshold. Specifically, for SantaCoder, the best performance is achieved with an edit distance of 60 and a repository inclusion threshold of 0.4. In contrast, both fine-tuned versions of Mistral and LLaMA-2 perform best with an edit distance threshold of 20. However, there are differences in their best-performing conditions: LLaMA-2 achieves a higher F-score with an inclusion threshold of 0.4, while Mistral performs better in detecting the inclusion of single scripts. Nonetheless, the performance gap between LLaMA-2 and Mistral in terms of dataset inclusion detection is not significant when using an edit distance threshold of 20. For all 3 models under study, we can observe that precision, accuracy, and F-score generally decrease as the edit distance threshold increases. This shows that a

higher threshold for edit distance may lead to less strict matching criteria, which in turn could result in more false positives and reduce *TraWiC*'s performance. Additionally, we observe that sensitivity marginally increases with the edit distance threshold for SantaCoder but marginally decreases for Mistral and LLaMA-2 when detecting single script inclusion. This suggests that while SantaCoder's classifier becomes less precise, it captures more true positives at higher thresholds. However, the performance of Mistral and LLaMA-2 degrades as the threshold increases.

Table 5.2 Results of *TraWiC*'s code inclusion detection with different edit distance thresholds on *SantaCoder*

Repository Inclusion Criterion	Edit Distance Threshold	Precision (%)	Accuracy (%)	F-score (%)	Sensitivity (%)	Specificity (%)
Considering only a **single** positive	**20**	88.07	83.87	88.57	89.08	71.60
Considering a threshold of **0.4**		76.52	75.10	84.82	95.13	20.63
Considering a threshold of **0.6**		80.46	71.26	80.31	80.15	47.08
Considering only a **single** positive	**40**	79.57	79.11	83.39	94.49	42.91
Considering a threshold of **0.4**		72.96	72.40	83.80	98.43	3.62
Considering a threshold of **0.6**		75.50	73.96	84.10	94.90	18.65
Considering only a **single** positive	**60**	78.09	77.46	85.46	94.36	37.70
Considering a threshold of **0.4**		71.28	71.23	82.95	99.19	2.92
Considering a threshold of **0.6**		73.30	72.11	82.89	95.38	15.60
Considering only a **single** positive	**70**	78.47	78.08	85.85	94.77	38.81
Considering a threshold of **0.4**		73.30	72.83	84.17	98.83	2.11
Considering a threshold of **0.6**		75.46	74.11	84.39	95.71	15.34
Considering only a **single** positive	**80**	77.79	77.47	85.55	95.02	36.17
Considering a threshold of **0.4**		71.22	70.83	82.84	99.00	1.47
Considering a threshold of **0.6**		72.42	70.98	82.41	95.60	10.30

Examining Table 5.2, we observe diverse results based on different edit distance thresholds and repository inclusion criteria. The highest F-score, 88.57%, is achieved at an edit distance threshold of 20 when considering a single script's inclusion. Specificity is highest, 71.6%, with the same criteria, indicating a decent balance in predicting when a script was **not** included in the model's training dataset. For project inclusion detection, we report a sensitivity of 99.19% and a specificity of 2.92% while considering an edit distance threshold of 60. This means that *TraWiC* has a high capability in detecting whether a project was included in the training dataset. As mentioned in Section 5.2.4, after following the data cleaning criteria as laid out in [125], some scripts from a project may not be included in the training dataset of the given model. For example, out of the 20 scripts in a project, 10 may not have a comment-to-code ratio between 1% to 80% and therefore will not be used for training the model. Therefore, even though the project itself was included in the model's training dataset, a specific script from it may not be. As such, it should be noted that most of the projects

in our dataset (in contrast to scripts) were included in SantaCoder's training dataset and a low specificity when it comes to project inclusion detection (as opposed to script inclusion detection) is not alarming as other scripts from the project could have been used in the model's training.

Table 5.3 Results of *TraWiC*'s code inclusion detection with different edit distance thresholds on *LLaMA-2*

Repository Inclusion Criterion	Edit Distance Threshold	Precision (%)	Accuracy (%)	F-score (%)	Sensitivity (%)	Specificity (%)
Considering only a **single** positive	**20**	78.68	82.74	81.36	84.23	81.54
Considering a threshold of **0.4**		80.00	86.27	82.05	84.21	87.50
Considering a threshold of **0.6**		75.00	72.54	68.18	62.50	81.48
Considering only a **single** positive	**40**	75.98	80.93	79.09	82.47	79.73
Considering a threshold of **0.4**		75.00	75.60	75.00	75.00	76.19
Considering a threshold of **0.6**		66.66	70.00	70.00	73.68	66.66
Considering only a **single** positive	**60**	73.49	79.06	76.56	79.91	78.43
Considering a threshold of **0.4**		70.00	73.17	71.79	73.68	72.72
Considering a threshold of **0.6**		70.00	70.73	70.00	70.00	71.42
Considering only a **single** positive	**70**	72.58	78.50	75.78	79.29	77.92
Considering a threshold of **0.4**		70.00	73.17	71.79	73.68	72.72
Considering a threshold of **0.6**		65.00	70.73	68.42	72.22	69.56
Considering only a **single** positive	**80**	70.56	77.57	74.46	78.82	76.67
Considering a threshold of **0.4**		75.00	68.29	69.76	65.21	72.22
Considering a threshold of **0.6**		70.00	63.41	65.11	60.89	66.66

Examining Tables 5.3 and 5.4, we observe various performance metrics for *TraWiC*'s performance on fine-tuned versions of LLaMA-2 and Mistral across different edit distance thresholds and repository inclusion criteria. For LLaMA-2 (Table 5.3), the highest F-score of 82.05% is achieved at an edit distance threshold of 20 when considering a repository inclusion criterion of 0.4. The highest specificity, 87.5%, is also achieved at an edit distance threshold of 20 with a repository inclusion criterion of 0.4. This indicates a good balance in predicting an entire project's inclusion in the model's training dataset. Sensitivity is highest at 84.23% with an edit distance threshold of 20 and considering only a single positive. This shows that at this edit distance threshold, *TraWiC* can detect both a single script's and a project's inclusion in a model's training dataset with high performance.

For Mistral (Table 5.4), the highest F-score, 85.37%, is achieved at an edit distance threshold of 20 when considering a single script's inclusion. Similarly, the highest sensitivity of 88.38% is observed under the same conditions. The highest specificity, 85.71%, is achieved with an edit distance threshold of 20 and a repository inclusion criterion of 0.4, indicating strong performance in predicting the absence of scripts in the training dataset. Therefore, we can observe that *TraWiC* shows high performance in code inclusion detection on LLMs trained on

Table 5.4 Results of *TraWiC*'s code inclusion detection with different edit distance thresholds on *Mistral*

Repository Inclusion Criterion	Edit Distance Threshold	Precision (%)	Accuracy (%)	F-score (%)	Sensitivity (%)	Specificity (%)
Considering only a **single** positive	**20**	82.55	86.45	85.37	88.38	84.89
Considering a threshold of **0.4**		85.00	85.36	85.00	85.00	85.71
Considering a threshold of **0.6**		84.21	82.50	82.05	80.00	85.00
Considering only a **single** positive	**40**	80.62	85.52	84.21	88.13	83.49
Considering a threshold of **0.4**		85.00	85.36	85.00	85.00	85.71
Considering a threshold of **0.6**		84.21	82.05	82.05	80.00	84.21
Considering only a **single** positive	**60**	77.51	83.48	81.79	86.58	81.16
Considering a threshold of **0.4**		80.00	80.48	80.00	80.00	80.95
Considering a threshold of **0.6**		80.00	78.04	78.04	76.19	80.00
Considering only a **single** positive	**70**	77.51	82.56	80.97	84.74	80.85
Considering a threshold of **0.4**		75.00	80.48	78.94	83.33	78.26
Considering a threshold of **0.6**		75.00	75.60	75.00	75.00	76.19
Considering only a **single** positive	**80**	74.74	83.11	81.31	86.46	80.64
Considering a threshold of **0.4**		65.00	70.73	68.42	72.22	69.56
Considering a threshold of **0.6**		65.00	63.41	63.41	61.90	65.00

code. It should also be noted that the results presented in Tables 5.3 and 5.4 are conducted on models that were fine-tuned on the dataset for 3 epochs. However, for both LLaMA-2 and Mistral models, we can see that *TraWiC*'s performance starts to degrade as we increase the edit distance threshold. We discuss the effects of fine-tuning and the reasons behind *TraWiC*'s performance degradation on these models in RQb_3.

Findings 1: When analyzing a single script's inclusion, *TraWiC* shows high performances in detecting both true positives and true negatives. The best balance is achieved when we employ lower edit distance thresholds for considering semantic identifiers. This means that looking for similarity in natural language elements of code can be used as a strong signal for detecting dataset inclusion.

Given the differences between *TraWiC*'s specificity and sensitivity on file-level granularity in comparison to repository-level granularity, we can infer two scenarios for using *TraWiC*. First, if a repository contains a large number of scripts and inference on the given model is resource-intensive, using *TraWiC* on less than half of the repository's scripts can effectively indicate dataset inclusion. Second, in scenarios where reducing false negatives is more important (e.g., identifying every instance of dataset inclusion), applying *TraWiC* on file-level granularity can be an effective method to check for dataset inclusion.

RQb$_1$-b: How does *TraWiC* compare against traditional CCD approaches for dataset inclusion detection?

Table 5.5 NiCad/JPlag's Performance on detecting clones across sampled repositories

Tool	Percentage of Dataset Sampled	Precision (%)	Accuracy (%)	F-Score (%)	Sensitivity (%)	Specificity (%)
NiCad	Sampling 5% (1726 projects / ≈ 34600 scripts)	84.81	43.13	53.60	38.18	63.63
	Sampling 9% (3108 projects / ≈ 62280 scripts)	86.74	47.64	56.47	41.86	72.50
	Sampling 10% (3452 projects / ≈ 69200 scripts)	80.24	41.22	49.24	35.51	64.44
JPlag	Sampling 5% (1726 projects / ≈ 34600 scripts)	83.53	54.49	58.92	45.51	77.26
	Sampling 9% (3108 projects / ≈ 62280 scripts)	82.63	53.83	58.32	45.06	76.02
	Sampling 10% (3452 projects / ≈ 69200 scripts)	81.00	55.00	59.94	47.57	73.00

Table 5.5 presents the results obtained when using NiCad and JPlag for clone detection. We used NiCad and JPlag to detect code clones across code snippets from the scripts in the LLM's training dataset and code snippets generated by the LLM as outlined in Section 5.3.6. Given that our task is a binary classification, NiCad is unable to achieve the accuracy of a random classifier (50%), and JPlag achieves an accuracy of 55%, which is barely higher than a random classifier indicating that code clone detection approaches are not suitable for detecting dataset inclusion. This observation is consistent with the results reported in [274].

The highest accuracy, F-Score, and sensitivity are achieved by NiCad for clone detection when sampling 9% of the LLM's dataset. The highest accuracy, F-score, and sensitivity for JPlag are achieved when sampling 10% of the dataset, while the highest precision and specificity are achieved when sampling 5% of the dataset. As the results show, our approach outperforms both NiCad and JPlag by a large margin in detecting whether a project was included in the training dataset of SantaCoder, regardless of the edit distance threshold. Moreover, using clone detection techniques for dataset inclusion detection is expensive. In fact, before using any clone detection technique, we need to generate code snippets using the LL,M and once a sufficient amount of scripts are generated, compare them in a pair-wise manner with the original scripts that were used in the LLM's training dataset. *TraWiC* is much less computationally expensive compared to generating multiple snippets of code from different repositories and using NiCad/JPlag to detect similarities between them. For detecting dataset inclusion, *TraWiC*'s main performance constraint is inference on the LLM for generating the necessary tokens. In contrast, both NiCad and JPlag need to compare an average of 3,000 scripts per project in a pair-wise manner for inclusion detection when considering a 10% sample of the filtered dataset on top of generating the necessary code from the LLM. As presented in Table 5.2, the best results are achieved when we focus on identifying the inclusion of single scripts with an edit distance threshold of 20. Our analysis

results presented in Table 5.5 show that increasing the number of sampled repositories has only a marginal improvement on NiCad's and JPlag's performance.

Findings 2: The high cost of pair-wise clone detection alongside the LLMs' capability in generating code with different syntax makes code clone detection tools incapable of detecting dataset inclusion. This is indicated by their poor performance, observed even when increasing the amount of code to compare.

RQb$_1$-c: What is the effect of using different classification methods on *TraWiC*'s performance?

Table 5.6 Performance comparison of different classifiers on the generated dataset

Edit Distance Threshold	Repository Inclusion Criterion	F-score XGB (%)	F-score SVM (%)	F-score RF (%)
20	Considering only a **single** positive	86.04	85.53	88.57
	Considering a threshold of **0.4**	83.53	82.30	84.82
	Considering a threshold of **0.6**	80.53	77.95	80.31
40	Considering only a **single** positive	84.01	83.20	83.39
	Considering a threshold of **0.4**	82.16	81.89	83.80
	Considering a threshold of **0.6**	82.48	80.60	84.10
60	Considering only a **single** positive	82.49	82.43	85.46
	Considering a threshold of **0.4**	82.18	82.20	82.95
	Considering a threshold of **0.6**	82.02	82.37	82.89
70	Considering only a **single** positive	84.32	84.49	85.85
	Considering a threshold of **0.4**	83.56	83.78	84.17
	Considering a threshold of **0.6**	81.85	82.82	84.39
80	Considering only a **single** positive	82.78	82.22	85.55
	Considering a threshold of **0.4**	82.39	81.91	82.84
	Considering a threshold of **0.6**	81.23	81.24	82.41

Table 5.6 shows the F-scores of different classification methods for SantaCoder. As the results show, random forests outperform both XGB and SVM across all edit distance thresholds and repository inclusion criteria.

5.4.2 RQb$_2$: *TraWiC*'s robustness against data obfuscation techniques

In this RQ, we assess *TraWiC*'s robustness to data obfuscation. Afterward, we analyze the correlation between the extracted features in our dataset and discuss the feature importance of the trained classifiers.

RQb$_2$-a: How robust is *TraWiC* against data obfuscation techniques?

In this section, we analyze *TraWiC*'s robustness against models trained on obfuscated datasets. Data obfuscation can be used as a defense strategy against MIAs [275]. For example, knowledge distillation [276, 277] or differential privacy [278, 279] can be used for training models that do not generate rare elements observed during their training. There also exists a myriad of code obfuscation techniques proposed for software copyright and vulnerability protection by making reverse-engineering the software more difficult. For example, Mixed Boolean Arithmetic methods [280] obfuscate the code by replacing arithmetic operations in a code with overly complex statements. Opaque Predicates approaches [281] obfuscate the code by setting only one direction of the code to be executed and inserting dummy code in the part that never gets executed, therefore, making the code more complex without introducing any change to the code's execution. Finally, other approaches alter identifiers, variables, and string literals or remove code indentation to change the code's semantics [282]. Approaches that use a combination of these methods [283] have been proposed for thorough code obfuscation to protect the code as well.

As such, in this section, we examine the robustness of our approach when confronted with models trained on obfuscated datasets. Such scenarios are also similar to detecting *Type-2/Type-3* code clones. It should be noted that for our sensitivity analysis, we assume that the data for training the given model has already been obfuscated before training the model, therefore making it more robust against producing textual elements similar to the original unobfuscated codes. To simulate these scenarios, we introduce noise during the hit-count process for syntactic identifiers. This noise emulates situations where the LLM produces a syntactic identifier that deviates from the original, resulting in a miss. Moreover, we test the robustness of our approach in scenarios where noise is introduced during the semantic identifiers' hit count. Finally, we simulate scenarios where both syntactic and semantic identifiers have been changed in the training dataset by data obfuscation techniques. Therefore, we will be able to test *TraWiC*'s robustness in recognizing dataset inclusion despite deliberate obfuscations in the training dataset.

Tables 5.7, 5.8, and 5.9 show the results of our sensitivity analysis conducted by varying the levels of noise (with noise ratios of 0.1, 0.5, and 0.9) injected during the syntactic, semantic, and their combination's hit-count process as mentioned above, for SantaCoder, LLaMA-2, and Mistral, respectively. Here, "Target" refers to the level that the noises are being applied to, a "Noise Ratio" of 0.1 means that during the hit-count process, a hit is ignored with a probability of 10%. Similar to the analyses done in RQb_1, we consider different edit distance thresholds (20, 60, and 80) for detecting semantic hits. It should be noted that we conduct

our analysis on single scripts instead of projects.

Table 5.7 Results of *TraWiC*'s dataset inclusion detection with noise injection across different levels on *SantaCoder*

Target	Edit Distance Threshold	Noise Ratio	Precision (%)	Accuracy (%)	F-score (%)	Sensitivity (%)	Specificity (%)
Syntactic	**20**	0.1	88.00	83.09	88.13	88.27	70.30
		0.5	86.41	79.26	85.23	84.08	67.34
		0.9	87.80	69.04	75.11	65.62	77.49
	60	0.1	79.45	78.51	86.18	94.17	39.85
		0.5	79.30	74.89	83.24	87.59	43.54
		0.9	81.82	59.73	66.37	55.83	69.37
	80	0.1	78.92	78.03	85.94	94.32	37.82
		0.5	78.43	74.95	83.55	89.39	39.30
		0.9	79.11	65.69	74.38	70.18	54.24
Semantic	**20**	0.1	87.19	82.29	87.62	88.04	68.08
		0.5	83.83	74.89	81.97	80.19	61.81
		0.9	81.70	51.86	55.22	41.70	76.94
	60	0.1	79.63	78.67	86.26	94.10	40.59
		0.5	78.83	76.33	84.59	91.26	39.48
		0.9	78.79	74.31	82.89	87.44	41.88
	80	0.1	79.31	78.88	86.51	95.14	38.75
		0.5	78.03	76.76	85.16	93.72	34.87
		0.9	78.13	75.21	83.86	90.51	37.45
Combined	**20**	0.1	87.16	82.07	87.45	87.74	68.08
		0.5	83.37	73.03	80.37	77.58	61.81
		0.9	85.25	50.96	52.18	37.59	83.95
	60	0.1	79.10	77.87	85.76	93.65	38.93
		0.5	78.93	73.51	82.15	85.65	43.54
		0.9	81.97	54.10	58.53	45.52	75.28
	80	0.1	78.34	77.39	85.59	94.32	35.61
		0.5	78.12	74.36	83.15	88.86	38.56
		0.9	82.10	57.45	63.24	51.42	72.32

As expected, increasing the noise sensitivity threshold results in a decline in precision, accuracy, F-score, and sensitivity across all edit distance thresholds, for all the models under study. This is aligned with the expectation that the introduction of noise impairs the classifier's ability to correctly identify true positives. However, the random forest's inherent robustness to overfitting and its ensemble nature appear to mitigate this effect to some extent, particularly at moderate levels of noise, as also reported in [284]. Finally, we observe that *TraWiC*'s performance degrades close to that of a random classifier when there exists a lot of noise in the dataset for SantaCoder and even further for LLaMA-2 and Mistral. This

indicates that our approach is not robust against thorough data obfuscation techniques where nearly all semantic and syntactic identifiers are changed.

Table 5.8 Results of *TraWiC*'s dataset inclusion detection with noise injection across different levels on *LLaMA-2*

Target	Edit Distance Threshold	Noise Ratio	Precision (%)	Accuracy (%)	F-score (%)	Sensitivity (%)	Specificity (%)
Syntactic	**20**	0.1	76.68	82.75	81.36	84.23	81.54
		0.5	68.60	71.80	69.96	71.37	72.16
		0.9	57.75	61.41	58.89	60.08	62.54
	60	0.1	72.29	76.26	73.92	75.63	76.77
		0.5	62.50	65.92	63.01	63.52	67.93
		0.9	56.22	59.25	56.22	56.22	61.99
	80	0.1	69.53	74.02	70.36	74.66	73.57
		0.5	60.48	66.54	62.63	64.96	67.76
		0.9	51.61	55.14	51.61	51.61	58.19
Semantic	**20**	0.1	70.93	74.40	72.62	74.39	74.40
		0.5	63.57	67.16	64.95	66.40	67.81
		0.9	53.10	56.40	53.83	54.58	57.99
	60	0.1	67.07	72.34	69.29	71.67	72.85
		0.5	57.43	62.43	58.73	60.08	64.31
		0.9	55.42	57.57	54.87	54.33	60.50
	80	0.1	66.13	73.46	69.79	73.87	73.16
		0.5	56.45	62.43	58.21	60.09	64.24
		0.9	49.60	54.39	50.20	50.83	57.43
Combined	**20**	0.1	68.60	71.43	69.69	70.80	71.97
		0.5	59.30	63.27	60.71	62.20	64.16
		0.9	53.10	55.66	53.41	53.73	57.39
	60	0.1	66.27	69.35	66.80	67.35	71.03
		0.5	59.04	64.49	60.74	62.55	66.00
		0.9	48.59	50.28	47.64	46.72	53.62
	80	0.1	62.50	68.60	64.85	67.39	69.51
		0.5	55.65	60.19	56.44	57.26	62.59
		0.9	47.18	54.02	48.75	50.43	56.77

We also conduct analyses of different "Repository Inclusion Criterion" (i.e., considering a project to be included in the model's training dataset if a certain amount of its corresponding scripts are detected as included in the model's training dataset) similar to the analysis done in Tables 5.2, 5.3, and 5.4, in Tables 5.10, 5.11, and 5.12.

Table 5.9 Results of *TraWiC*'s dataset inclusion detection with noise injection across different levels on *Mistral*

Target	Edit Distance Threshold	Noise Ratio	Precision (%)	Accuracy (%)	F-score (%)	Sensitivity (%)	Specificity (%)
Syntactic	**20**	0.1	82.56	86.48	85.37	88.38	84.95
		0.5	71.32	73.15	71.73	72.16	74.04
		0.9	66.15	66.85	65.24	65.37	68.20
	60	0.1	77.52	83.49	81.80	86.58	81.17
		0.5	67.83	71.61	69.58	71.43	71.77
		0.9	55.43	61.22	57.78	60.34	61.92
	80	0.1	73.26	78.66	76.67	80.43	77.30
		0.5	64.34	71.24	68.17	72.49	70.32
		0.9	55.42	60.75	57.02	58.72	62.37
Semantic	**20**	0.1	72.48	76.44	74.65	76.95	76.01
		0.5	67.05	69.02	67.45	67.84	70.07
		0.9	56.98	60.11	57.76	58.57	61.46
	60	0.1	68.99	74.58	72.21	75.74	73.68
		0.5	61.63	66.42	63.73	65.98	66.78
		0.9	56.59	60.30	57.71	58.87	61.51
	80	0.1	67.44	74.03	71.31	75.65	72.82
		0.5	59.69	67.16	63.51	67.84	66.67
		0.9	53.49	59.55	55.87	58.47	60.40
Combined	**20**	0.1	72.48	75.88	74.21	76.02	75.77
		0.5	65.89	69.02	67.06	68.27	69.66
		0.9	58.91	61.78	59.61	69.32	63.07
	60	0.1	69.77	72.36	70.73	71.71	72.92
		0.5	60.85	64.38	62.06	63.31	65.29
		0.9	53.88	54.92	53.36	52.85	56.88
	80	0.1	67.05	72.73	70.18	73.62	72.04
		0.5	61.24	64.75	62.45	63.71	65.64
		0.9	52.71	57.88	54.51	56.43	59.06

Cross-analyzing the results of Tables 5.7, 5.8, and 5.9 with Tables 5.2, 5.3, and 5.4, which contain the classifier's performance on clean data, provides the following observations:

- **Semantic identifiers are important in dataset inclusion detection**: We observe sharp degradation in the classifier's performance at higher edit distance thresholds. As

shown in Figures 5.5 and 5.3 which display the feature importance of our trained classifiers, we can observe that at lower edit distance thresholds, the classifier gives more weight to semantic identifiers compared to syntactic identifiers for all of the 3 models under study. Therefore, at lower edit distance thresholds, even though we observe performance degradation, the classifier has a high performance in detecting true positives even as the amount of noise is increased. As mentioned in Chapter 2.3.2, *Type-2* code clones are similar to each other in syntax but have different variable/function/class names. Our results on simulating data obfuscation show that *TraWiC* can successfully detect dataset inclusion despite changing the syntactic identifiers.

- **Considering a repository-level granularity for dataset inclusion detection can help against data obfuscation**: Comparing Tables 5.2, 5.3, and 5.4 with Tables 5.10, 5.11, and 5.12, we can observe degradation in detecting dataset inclusion in file-level granularity on higher edit distance thresholds. This decrease in performance is indicated by the decrease in sensitivity (correctly classified true positives) and the increase in specificity (correctly classified true negatives). As mentioned above, as a large number of the scripts in our constructed dataset have been included in the training dataset of the model under study, we cannot attribute the rise in specificity as an indicator of *TraWiC*'s better performance for SantaCoder, and as we can observe from Tables 5.8 and 5.9, for both Mistral and LLaMA the performance of the classifier degrades as the amount of noise is increased. However, even though we observe degradation in performance while considering repository-level granularity as well, we observe that analyzing all the scripts in a project can help us detect dataset inclusion even in the presence of high levels of noise for all three models under study. Depending on the approach used for code obfuscation, obfuscating the entire project is either too costly or adds unnecessary performance overhead to the project's operation [282,285,286] and many of the proposed approaches focus on obfuscating blocks of code inside the script as opposed to its entirety [283,287]. As such, even though the identifiers in one script in the project may be changed, other scripts in the same project may remain unchanged or undergo lower amounts of obfuscation, therefore aiding in dataset inclusion detection.

Table 5.10 Results of *TraWiC*'s dataset inclusion detection with noise injection across different levels on *SantaCoder* - **Repository level**

Target	Edit Distance Threshold	Noise Ratio	Repository Inclusion Criterion	Precision (%)	Accuracy (%)	F-score (%)	Sensitivity (%)	Specificity (%)
Semantic Identifiers	**20**		Considering only a **single** positive	87.19	82.29	87.62	88.04	68.08
		0.1	Considering a threshold of **0.4**	79.56	76.82	86.17	93.97	20.00
			Considering a threshold of **0.6**	84.55	73.51	82.30	80.17	51.43
			Considering only a **single** positive	83.83	74.89	81.97	80.19	61.81
		0.5	Considering a threshold of **0.4**	79.14	76.82	86.27	94.83	17.14
			Considering a threshold of **0.6**	85.32	74.17	82.67	80.17	54.29
			Considering only a **single** positive	81.70	51.86	55.22	41.70	76.94
		0.9	Considering a threshold of **0.4**	80.15	75.50	85.02	90.52	25.71
			Considering a threshold of **0.6**	87.18	61.59	70.10	58.62	71.43
	40		Considering only a **single** positive	80.47	79.79	86.94	94.54	43.36
		0.1	Considering a threshold of **0.4**	77.70	77.48	87.12	99.14	5.71
			Considering a threshold of **0.6**	81.29	80.79	88.63	97.41	25.71
			Considering only a **single** positive	79.07	76.86	84.95	91.78	40.04
		0.5	Considering a threshold of **0.4**	77.18	76.82	86.79	99.14	2.86
			Considering a threshold of **0.6**	81.16	80.13	88.19	96.55	25.71
			Considering only a **single** positive	77.96	70.27	79.53	81.17	43.36
		0.9	Considering a threshold of **0.4**	78.23	78.15	87.45	99.14	8.57
			Considering a threshold of **0.6**	81.62	80.13	88.10	95.69	28.57
	80		Considering only a **single** positive	79.31	78.88	86.51	95.14	38.75
		0.1	Considering a threshold of **0.4**	77.70	77.48	87.12	99.14	5.71
			Considering a threshold of **0.6**	80.71	80.13	88.28	97.41	22.86
			Considering only a **single** positive	78.03	76.76	85.16	93.72	34.87
		0.5	Considering a threshold of **0.4**	77.33	77.48	87.22	100.00	2.86
			Considering a threshold of **0.6**	80.71	80.13	88.28	97.41	22.86
			Considering only a **single** positive	78.13	75.21	83.86	90.51	37.45
		0.9	Considering a threshold of **0.4**	77.85	78.15	87.55	100.00	5.71
			Considering a threshold of **0.6**	81.43	81.46	89.06	98.28	25.71
Syntactic Identifiers	**20**		Considering only a **single** positive	88.00	83.09	88.13	88.27	70.30
		0.1	Considering a threshold of **0.4**	79.56	76.82	86.17	93.97	20.00
			Considering a threshold of **0.6**	80.73	67.55	78.22	75.86	40.00
			Considering only a **single** positive	86.41	79.26	85.23	84.08	67.34
		0.5	Considering a threshold of **0.4**	78.68	74.83	84.92	92.24	17.14
			Considering a threshold of **0.6**	81.13	66.89	77.48	74.14	42.86
			Considering only a **single** positive	87.80	69.04	75.11	65.62	77.49
		0.9	Considering a threshold of **0.4**	78.52	74.17	84.46	91.38	17.14
			Considering a threshold of **0.6**	83.51	66.23	76.06	69.83	54.29
	40		Considering only a **single** positive	80.34	79.41	86.68	94.10	43.17
		0.1	Considering a threshold of **0.4**	78.23	78.15	87.45	99.14	8.57
			Considering a threshold of **0.6**	81.16	80.13	88.19	96.55	25.71
			Considering only a **single** positive	79.52	75.00	83.26	87.37	44.46
		0.5	Considering a threshold of **0.4**	77.70	77.48	87.12	99.14	5.71
			Considering a threshold of **0.6**	81.48	79.47	87.65	94.83	28.57
			Considering only a **single** positive	83.74	64.15	70.97	61.58	70.48
		0.9	Considering a threshold of **0.4**	77.93	76.82	86.59	97.41	8.57
			Considering a threshold of **0.6**	83.61	77.48	85.71	87.93	42.86
	80		Considering only a **single** positive	78.92	78.03	85.94	94.32	37.82
		0.1	Considering a threshold of **0.4**	77.85	78.15	87.55	100.00	5.71
			Considering a threshold of **0.6**	80.71	80.13	88.28	97.41	22.86
			Considering only a **single** positive	78.43	74.95	83.55	89.39	39.30
		0.5	Considering a threshold of **0.4**	77.33	77.48	87.22	100.00	2.86
			Considering a threshold of **0.6**	82.01	82.12	89.41	98.28	28.57
			Considering only a **single** positive	79.11	65.59	74.38	70.18	54.24
		0.9	Considering a threshold of **0.4**	78.38	78.81	87.88	100.00	8.57
			Considering a threshold of **0.6**	82.71	80.79	88.35	94.83	34.29
Combined Identifiers	**20**		Considering only a **single** positive	87.16	82.07	87.45	87.74	68.08
		0.1	Considering a threshold of **0.4**	79.71	77.48	86.61	94.83	20.00
			Considering a threshold of **0.6**	83.93	73.51	82.46	81.03	48.57
			Considering only a **single** positive	83.37	73.03	80.37	77.58	61.81
		0.5	Considering a threshold of **0.4**	79.71	77.48	86.61	94.83	20.00
			Considering a threshold of **0.6**	85.45	74.83	83.19	81.03	54.29
			Considering only a **single** positive	85.25	50.96	52.18	37.59	83.95
		0.9	Considering a threshold of **0.4**	82.11	75.50	84.52	87.07	37.14
			Considering a threshold of **0.6**	87.84	60.26	68.42	56.03	74.29
	40		Considering only a **single** positive	79.51	78.30	86.00	93.65	40.41
		0.1	Considering a threshold of **0.4**	78.23	78.15	87.45	99.14	8.57
			Considering a threshold of **0.6**	80.71	80.13	88.28	97.41	22.86
			Considering only a **single** positive	78.94	73.99	82.57	86.55	42.99
		0.5	Considering a threshold of **0.4**	77.70	77.48	87.12	99.14	5.71
			Considering a threshold of **0.6**	81.16	80.13	88.19	96.55	25.71
			Considering only a **single** positive	84.54	55.21	59.05	45.37	79.52
		0.9	Considering a threshold of **0.4**	79.43	78.15	87.16	96.55	17.14
			Considering a threshold of **0.6**	80.95	66.23	76.92	73.28	42.86
	80		Considering only a **single** positive	78.34	77.39	85.59	94.32	35.61
		0.1	Considering a threshold of **0.4**	77.70	77.48	87.12	99.14	5.71
			Considering a threshold of **0.6**	82.01	82.12	89.41	98.28	28.57
			Considering only a **single** positive	78.12	74.36	83.15	88.86	38.56
		0.5	Considering a threshold of **0.4**	77.70	77.48	87.12	99.14	5.71
			Considering a threshold of **0.6**	81.29	80.79	88.63	97.41	25.71
			Considering only a **single** positive	82.10	57.45	63.24	51.42	72.32
		0.9	Considering a threshold of **0.4**	79.17	78.81	87.69	98.28	14.29
			Considering a threshold of **0.6**	82.20	73.51	82.91	83.62	40.00

Table 5.11 Results of *TraWiC*'s dataset inclusion detection with noise injection across different levels on *LLaMA-2* - **Repository level**

Target	Edit Distance Threshold	Noise Ratio	Repository Inclusion Criterion	Precision (%)	Accuracy (%)	F-score (%)	Sensitivity (%)	Specificity (%)
Semantic Identifiers	**20**	0.1	Considering only a **single** positive	70.93	74.40	72.62	74.39	74.40
			Considering a threshold of **0.4**	75.00	80.49	78.95	83.33	78.26
			Considering a threshold of **0.6**	60.00	64.71	57.14	54.55	72.41
		0.5	Considering only a **single** positive	63.57	67.16	64.95	66.40	67.81
			Considering a threshold of **0.4**	70.00	73.17	71.79	73.68	72.73
			Considering a threshold of **0.6**	60.00	60.78	54.55	50.00	70.37
		0.9	Considering only a **single** positive	53.10	56.40	53.83	54.58	57.99
			Considering a threshold of **0.4**	70.00	60.98	63.64	58.33	64.71
			Considering a threshold of **0.6**	50.00	47.06	42.55	37.04	58.33
	40	0.1	Considering only a **single** positive	69.29	72.15	70.26	71.26	72.92
			Considering a threshold of **0.4**	70.00	68.29	68.29	66.67	70.00
			Considering a threshold of **0.6**	61.90	67.50	66.67	72.22	63.64
		0.5	Considering only a **single** positive	78.94	73.99	82.57	86.55	42.99
			Considering a threshold of **0.4**	75.98	80.93	79.10	82.48	79.73
			Considering a threshold of **0.6**	52.38	60.00	57.89	64.71	56.52
		0.9	Considering only a **single** positive	52.76	56.07	53.28	53.82	58.04
			Considering a threshold of **0.4**	65.00	60.98	61.90	59.09	63.16
			Considering a threshold of **0.6**	42.86	52.50	48.65	56.25	50.00
	80	0.1	Considering only a **single** positive	66.13	73.46	69.79	73.87	73.16
			Considering a threshold of **0.4**	70.00	63.41	65.12	60.87	66.67
			Considering a threshold of **0.6**	65.00	58.54	60.47	56.52	61.11
		0.5	Considering only a **single** positive	56.45	62.43	58.21	60.09	64.24
			Considering a threshold of **0.4**	65.00	56.10	59.09	54.17	58.82
			Considering a threshold of **0.6**	55.00	48.78	51.16	47.83	50.00
		0.9	Considering only a **single** positive	49.60	54.39	50.20	50.83	57.43
			Considering a threshold of **0.4**	50.00	48.78	48.78	47.62	50.00
			Considering a threshold of **0.6**	50.00	43.90	46.51	43.48	44.44
Syntactic Identifiers	**20**	0.1	Considering only a **single** positive	76.68	82.57	81.36	84.23	81.54
			Considering a threshold of **0.4**	80.00	82.93	82.05	84.21	81.82
			Considering a threshold of **0.6**	70.00	66.67	62.22	56.00	76.92
		0.5	Considering only a **single** positive	68.60	71.80	69.96	71.37	72.16
			Considering a threshold of **0.4**	70.00	73.17	71.79	73.68	72.73
			Considering a threshold of **0.6**	70.00	64.71	60.87	53.85	76.00
		0.9	Considering only a **single** positive	57.75	61.41	58.89	60.08	62.54
			Considering a threshold of **0.4**	60.00	60.98	60.00	60.00	61.90
			Considering a threshold of **0.6**	60.00	56.86	52.17	46.15	68.00
	40	0.1	Considering only a **single** positive	73.23	77.01	75.15	77.18	76.87
			Considering a threshold of **0.4**	75.00	75.61	75.00	75.00	76.19
			Considering a threshold of **0.6**	61.90	65.00	65.00	68.42	61.90
		0.5	Considering only a **single** positive	57.09	59.44	57.20	57.31	61.37
			Considering a threshold of **0.4**	75.00	75.61	75.00	75.00	76.19
			Considering a threshold of **0.6**	57.14	55.00	57.14	57.14	52.63
		0.9	Considering only a **single** positive	43.31	45.05	42.80	42.31	47.64
			Considering a threshold of **0.4**	70.00	60.98	63.64	58.33	64.71
			Considering a threshold of **0.6**	47.62	57.50	54.05	62.50	54.17
	80	0.1	Considering only a **single** positive	69.53	74.02	70.36	74.66	73.57
			Considering a threshold of **0.4**	75.00	65.85	68.18	62.50	70.59
			Considering a threshold of **0.6**	65.00	60.98	61.90	59.09	63.16
		0.5	Considering only a **single** positive	60.48	66.54	62.63	64.96	67.76
			Considering a threshold of **0.4**	60.00	56.10	57.14	54.55	57.89
			Considering a threshold of **0.6**	65.00	60.98	61.90	59.09	63.16
		0.9	Considering only a **single** positive	51.61	55.14	51.61	51.61	58.19
			Considering a threshold of **0.4**	60.00	53.66	55.81	52.17	55.56
			Considering a threshold of **0.6**	60.00	53.66	55.81	52.17	55.56
Combined Identifiers	**20**	0.1	Considering only a **single** positive	68.60	71.43	69.69	70.80	71.97
			Considering a threshold of **0.4**	75.00	75.61	75.00	75.00	76.19
			Considering a threshold of **0.6**	65.00	62.75	57.78	52.00	73.08
		0.5	Considering only a **single** positive	59.30	63.27	60.71	62.20	64.16
			Considering a threshold of **0.4**	65.00	68.29	66.67	68.42	68.18
			Considering a threshold of **0.6**	55.00	56.86	50.00	45.83	66.67
		0.9	Considering only a **single** positive	53.10	55.66	53.41	53.73	57.39
			Considering a threshold of **0.4**	55.00	46.34	50.00	45.83	47.06
			Considering a threshold of **0.6**	45.00	47.06	40.00	36.00	57.69
	40	0.1	Considering only a **single** positive	60.24	63.74	61.20	62.20	65.05
			Considering a threshold of **0.4**	65.00	68.29	66.67	68.42	68.18
			Considering a threshold of **0.6**	61.90	60.00	61.90	61.90	57.89
		0.5	Considering only a **single** positive	57.09	61.50	58.47	59.92	62.80
			Considering a threshold of **0.4**	50.00	53.66	51.28	52.63	54.55
			Considering a threshold of **0.6**	52.38	55.00	55.00	57.89	52.38
		0.9	Considering only a **single** positive	54.33	58.69	55.53	56.79	60.27
			Considering a threshold of **0.4**	60.00	48.78	53.33	48.00	50.00
			Considering a threshold of **0.6**	45.00	47.06	40.00	36.00	57.69
	80	0.1	Considering only a **single** positive	62.50	68.60	64.85	67.39	69.51
			Considering a threshold of **0.4**	65.00	63.41	63.41	61.90	65.00
			Considering a threshold of **0.6**	65.00	56.10	59.09	54.17	58.82
		0.5	Considering only a **single** positive	55.65	60.19	56.44	57.26	62.59
			Considering a threshold of **0.4**	55.00	56.10	55.00	55.00	57.14
			Considering a threshold of **0.6**	50.00	48.78	48.78	47.62	50.00
		0.9	Considering only a **single** positive	47.18	54.02	48.75	50.43	56.77
			Considering a threshold of **0.4**	45.00	43.90	43.90	42.86	45.00
			Considering a threshold of **0.6**	45.00	48.78	46.15	47.37	50.00

Table 5.12 Results of *TraWiC*'s dataset inclusion detection with noise injection across different levels on *Mistral* - **Repository level**

Target	Edit Distance Threshold	Noise Ratio	Repository Inclusion Criterion	Precision (%)	Accuracy (%)	F-score (%)	Sensitivity (%)	Specificity (%)
Semantic Identifiers	**20**	0.1	Considering only a **single** positive	72.48	76.44	74.65	76.95	76.0
			Considering a threshold of **0.4**	75.00	78.05	76.92	78.95	77.27
			Considering a threshold of **0.6**	68.42	70.00	68.42	68.42	71.43
		0.5	Considering only a **single** positive	67.05	69.02	67.45	67.84	70.07
			Considering a threshold of **0.4**	75.00	70.73	71.43	68.18	73.68
			Considering a threshold of **0.6**	68.42	65.00	65.00	61.90	68.42
		0.9	Considering only a **single** positive	56.98	60.11	57.76	58.57	62.37
			Considering a threshold of **0.4**	60.00	56.10	57.14	54.55	57.89
			Considering a threshold of **0.6**	73.68	60.00	63.64	56.00	66.67
	40	0.1	Considering only a **single** positive	80.25	79.77	78.44	76.71	82.59
			Considering a threshold of **0.4**	85.00	82.93	82.93	80.95	85.00
			Considering a threshold of **0.6**	68.42	71.79	70.27	72.22	71.43
		0.5	Considering only a **single** positive	68.07	69.17	66.94	65.85	72.16
			Considering a threshold of **0.4**	70.00	75.61	73.68	77.78	73.91
			Considering a threshold of **0.6**	73.68	69.23	70.00	66.67	72.22
		0.9	Considering only a **single** positive	58.82	60.50	57.73	56.68	63.97
			Considering a threshold of **0.4**	50.00	63.41	57.14	66.67	61.54
			Considering a threshold of **0.6**	63.16	51.28	55.81	50.00	53.33
	80	0.1	Considering only a **single** positive	67.44	74.03	71.31	75.65	72.82
			Considering a threshold of **0.4**	65.00	68.29	66.67	68.42	68.18
			Considering a threshold of **0.6**	60.00	58.54	58.54	57.14	60.00
		0.5	Considering only a **single** positive	59.69	67.16	63.51	67.84	66.67
			Considering a threshold of **0.4**	50.00	56.10	52.63	55.56	56.52
			Considering a threshold of **0.6**	45.00	48.78	46.15	47.37	50.00
		0.9	Considering only a **single** positive	53.49	59.55	55.87	58.47	60.40
			Considering a threshold of **0.4**	50.00	60.98	55.56	62.50	60.00
			Considering a threshold of **0.6**	60.00	51.22	54.55	50.00	52.94
Syntactic Identifiers	**20**	0.1	Considering only a **single** positive	82.56	86.48	85.37	88.38	84.95
			Considering a threshold of **0.4**	85.00	82.93	82.93	80.95	85.00
			Considering a threshold of **0.6**	84.21	82.50	82.50	80.00	85.00
		0.5	Considering only a **single** positive	71.32	73.15	71.73	72.16	74.04
			Considering a threshold of **0.4**	80.00	78.05	78.05	76.19	80.00
			Considering a threshold of **0.6**	73.68	72.50	71.79	70.00	75.00
		0.9	Considering only a **single** positive	66.15	66.85	65.24	65.37	68.20
			Considering a threshold of **0.4**	75.00	65.85	68.18	62.50	70.59
			Considering a threshold of **0.6**	73.68	70.00	70.00	66.67	73.68
	40	0.1	Considering only a **single** positive	78.29	80.71	79.53	80.80	80.62
			Considering a threshold of **0.4**	75.00	78.05	76.92	78.95	77.27
			Considering a threshold of **0.6**	73.68	71.79	71.79	70.00	73.68
		0.5	Considering only a **single** positive	70.93	74.77	72.91	75.00	74.58
			Considering a threshold of **0.4**	60.00	63.41	61.54	63.16	63.64
			Considering a threshold of **0.6**	68.42	69.23	68.42	68.42	70.00
		0.9	Considering only a **single** positive	63.57	65.86	64.06	64.57	67.02
			Considering a threshold of **0.4**	55.00	60.98	57.89	61.11	60.87
			Considering a threshold of **0.6**	52.63	56.41	54.05	55.56	57.14
	80	0.1	Considering only a **single** positive	73.26	78.66	76.67	80.43	77.30
			Considering a threshold of **0.4**	65.00	70.73	68.42	72.22	69.57
			Considering a threshold of **0.6**	60.00	56.10	57.14	54.55	57.89
		0.5	Considering only a **single** positive	64.34	71.24	68.17	72.49	70.32
			Considering a threshold of **0.4**	55.00	58.54	56.41	57.89	59.09
			Considering a threshold of **0.6**	55.00	51.22	52.38	50.00	52.63
		0.9	Considering only a **single** positive	55.42	60.75	57.02	58.72	62.37
			Considering a threshold of **0.4**	50.00	48.78	48.78	47.62	50.00
			Considering a threshold of **0.6**	50.00	51.22	50.00	50.00	52.38
Combined Identifiers	**20**	0.1	Considering only a **single** positive	72.48	75.88	74.21	76.02	75.77
			Considering a threshold of **0.4**	75.00	75.61	75.00	75.00	76.19
			Considering a threshold of **0.6**	84.21	80.00	80.00	76.19	84.21
		0.5	Considering only a **single** positive	65.89	69.02	67.06	68.27	69.66
			Considering a threshold of **0.4**	70.00	73.17	71.79	73.68	72.73
			Considering a threshold of **0.6**	63.16	60.00	60.00	57.14	63.16
		0.9	Considering only a **single** positive	58.91	61.78	59.61	69.32	63.07
			Considering a threshold of **0.4**	55.00	53.66	53.66	52.38	55.00
			Considering a threshold of **0.6**	57.89	55.00	55.00	52.38	57.89
	40	0.1	Considering only a **single** positive	69.77	73.84	71.86	74.07	73.65
			Considering a threshold of **0.4**	75.00	78.05	76.92	78.95	77.27
			Considering a threshold of **0.6**	78.95	74.36	75.00	71.43	77.78
		0.5	Considering only a **single** positive	63.18	66.05	64.05	64.94	67.01
			Considering a threshold of **0.4**	70.00	70.73	70.00	70.00	71.43
			Considering a threshold of **0.6**	73.68	66.67	68.29	63.64	70.59
		0.9	Considering only a **single** positive	51.55	58.63	54.40	57.58	59.42
			Considering a threshold of **0.4**	55.00	58.54	56.41	57.89	59.09
			Considering a threshold of **0.6**	73.68	66.67	68.29	63.64	70.59
	80	0.1	Considering only a **single** positive	67.05	72.73	70.18	73.62	72.04
			Considering a threshold of **0.4**	65.00	65.85	65.00	65.00	66.67
			Considering a threshold of **0.6**	60.00	58.54	58.54	57.14	60.00
		0.5	Considering only a **single** positive	61.24	64.75	62.45	63.71	65.64
			Considering a threshold of **0.4**	50.00	56.10	52.63	55.56	56.52
			Considering a threshold of **0.6**	55.00	53.66	53.66	52.38	55.00
		0.9	Considering only a **single** positive	52.71	57.88	54.51	56.43	59.06
			Considering a threshold of **0.4**	40.00	39.02	39.02	38.10	40.00
			Considering a threshold of **0.6**	40.00	41.46	40.00	40.00	42.86

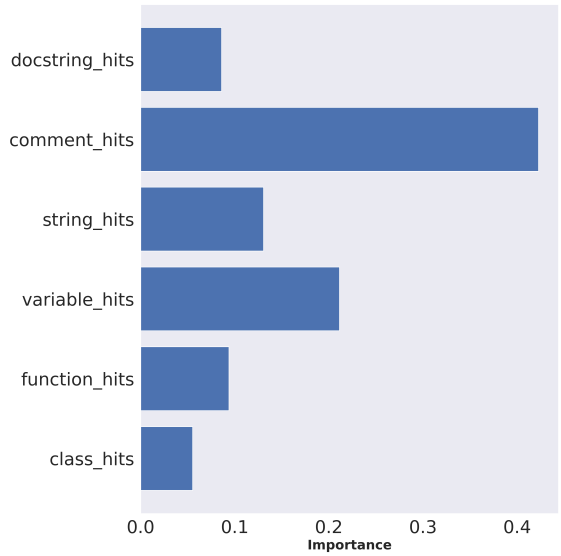
Findings 3: Data obfuscation techniques can be used to prevent MIAs from detecting dataset inclusion. Our results show that *TraWiC* is capable of detecting dataset inclusion when a moderate level of noise is applied to the training data. Moreover, we show that by analyzing all the scripts in a project, we can detect dataset inclusion with an F-score of up to 83.15%, even when half of the semantic and syntactic identifiers in code are obfuscated.

RQb$_2$-b: What is the importance of each feature in detecting dataset inclusion?

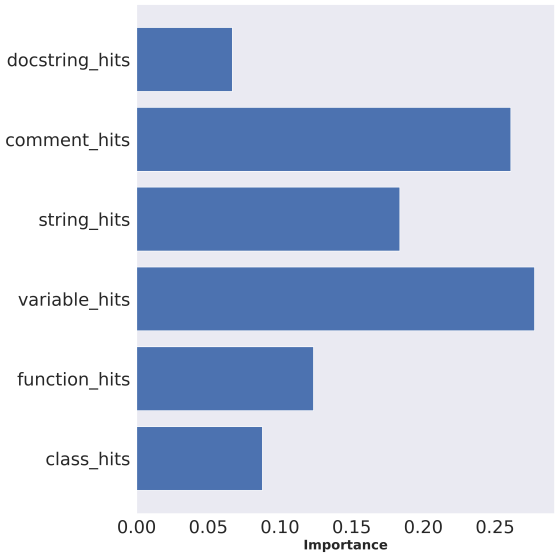
We first analyze the constructed dataset for training the classification models using Spearman's rank correlation coefficient [288] to assess the relationship between the constructed features of the dataset. Then, we analyze the classification models' feature importance using the Gini importance criterion [289].

Figure 5.4 displays the correlation analysis of the extracted features with `trained_on` being the dependent variable indicating a script's inclusion in the LLM's training dataset for each of the models under study. We observe weak to no correlation between different features, which indicates low dependency among features and that each feature provides unique information for the classification task. As displayed in Tables 5.2, 5.3, and 5.4, we experiment with multiple edit distance thresholds for counting semantic hits. Even though the performances of models for different edit distance thresholds are similar, there are variations in the classifiers' feature importance. Figure 5.3 displays different feature importance distributions for detecting dataset inclusion for SantaCoder for edit distances of 20, 40, 60, and 80, respectively. As displayed in this figure, the lower the edit distance threshold is, the more the importance of *docstring* and *comment* features for deciding code inclusion increases. Comments and docstrings are specifically designed to explain the code's functionality, as they are written to explain pieces of code in natural language, which makes them inherently unique. Therefore, with lower edit distances, which lowers the matching criterion between the original tokens and what the model generates, the number of hits for semantic elements increases, and as a result, the importance of these features increases.

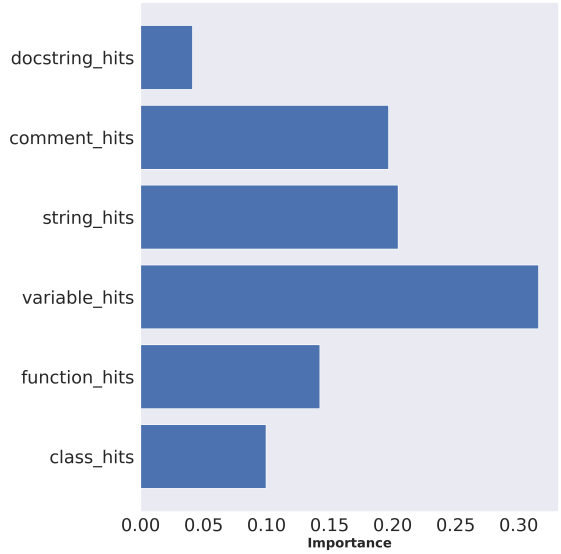
Figure 5.3c displays the feature importance of the best-performing classifier (i.e., the random forest with an edit distance of 60 and repository inclusion criterion of 0.4) for SantaCoder. The number of variable hits is the most significant feature, followed by the number of comment hits and function hits. The number of docstring hits has the least importance, which suggests that the information within the docstrings of the code has a minimal impact on the classifier's decisions. It should be noted that, compared to other semantic identifiers,



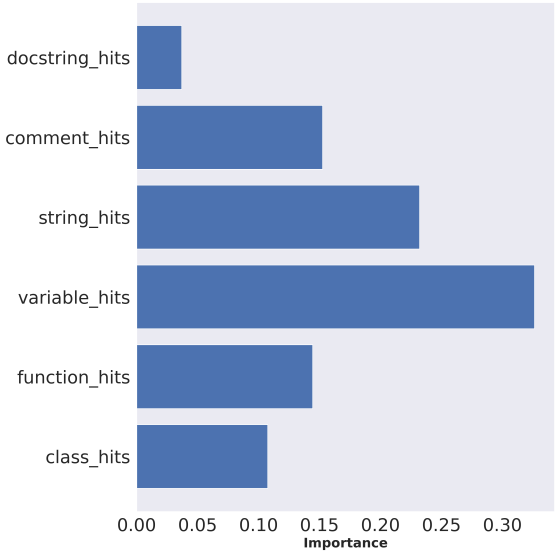
(a) Feature Importance (edit distance of 20)



(b) Feature Importance (edit distance of 40)

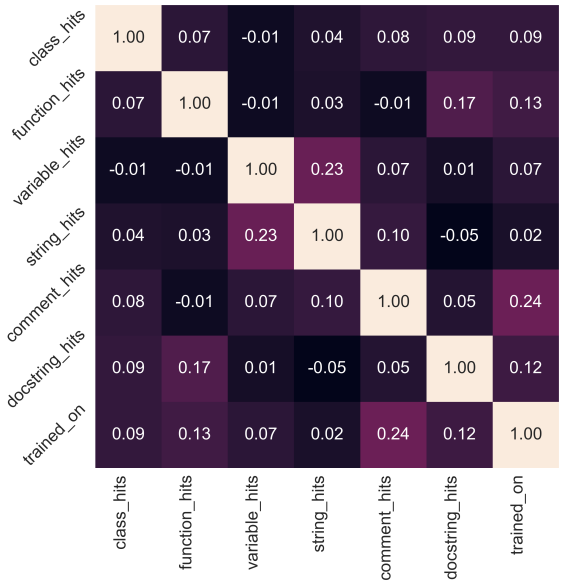


(c) Feature Importance (edit distance of 60)

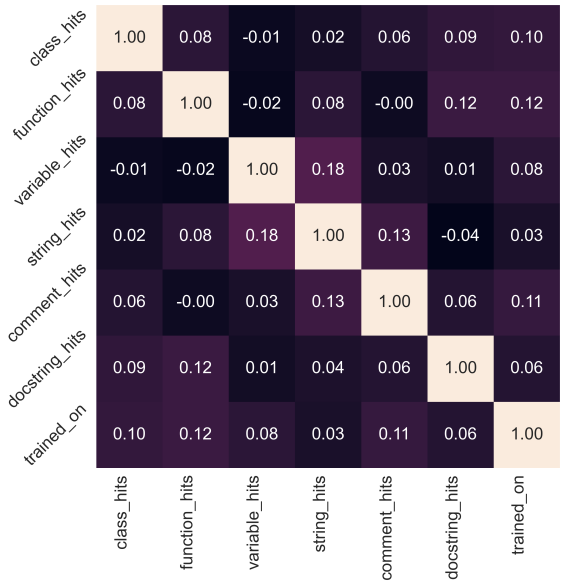


(d) Feature Importance (edit distance of 80)

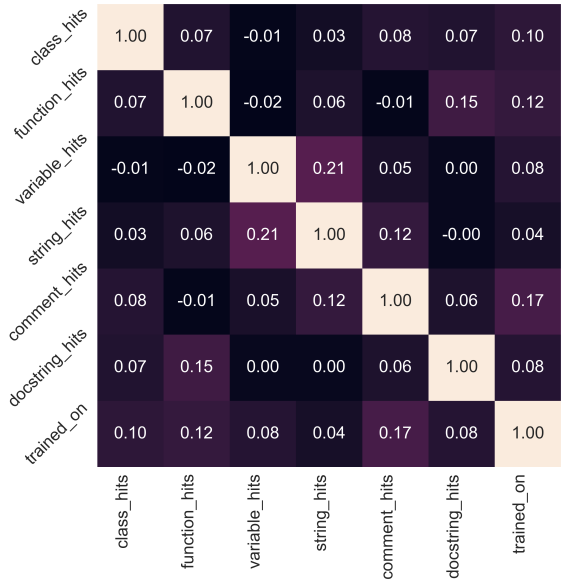
Figure 5.3 Feature importance of the final dataset with different edit distance thresholds on *SantaCoder*



(a) SantaCoder



(b) LLaMA-2



(c) Mistral

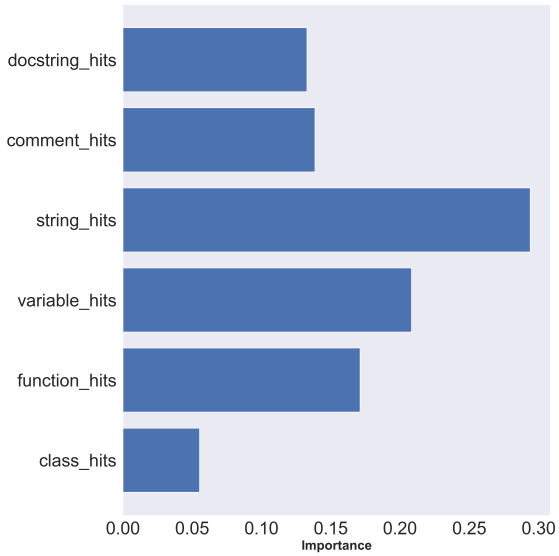
Figure 5.4 Correlation of different features in the generated dataset across *SantaCoder*, *LLaMA-2*, and *Mistral*

docstrings are longer (as they explain the functionality of a class alongside its inputs and outputs) and less prevalent (one per function/class). Therefore, there exists a smaller number of hits for docstrings compared to the other semantic identifiers.

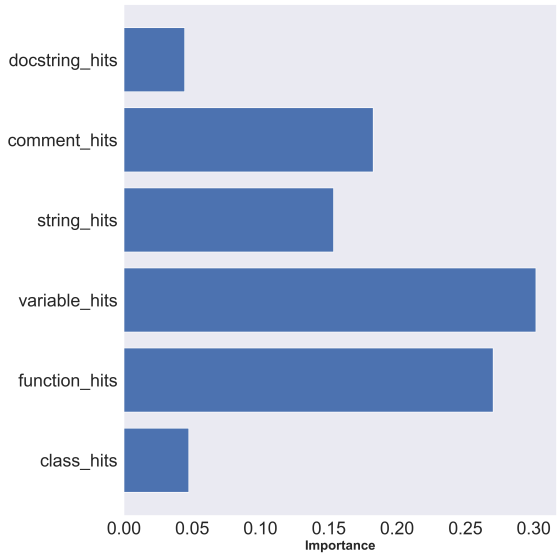
We observe different feature importance and correlations of features for the fine-tuned models of Mistral and LLaMA compared to SantaCoder, as displayed in Figure 5.5. Figure 5.5a shows that for an edit distance of 20 for LLaMA-2, the number of string hits is the most significant feature, followed by variable hits and function hits. In Figure 5.5c, which presents the feature importance for an edit distance of 60 for LLaMA-2, the number of variable hits becomes the most significant feature, followed by comment hits and string hits. It should be noted that the feature importance of comment, string, and function hits is relatively the same at this edit distance. However, at both edit distances, the number of class hits has the least importance (excluding the docstring hits for an edit distance of 60). This is due to the relatively fewer occurrences of classes in scripts within the analyzed dataset, which is similar to the analysis carried out for SantaCoder. Figure 5.5b and 5.5d display the feature importance of the classifier at edit distances of 20 and 60 for Mistral, respectively. At an edit distance of 20, the number of variable hits is the most significant feature, followed by function hits and comment hits. The number of class hits is the least important, suggesting that the information within class definitions has minimal impact on the classifier's decisions, similar to LLaMA-2 and SantaCoder. At an edit distance of 60, the number of variable hits remains the most significant feature, followed by string hits and comment hits. The consistency across different semantic thresholds for all three models shows that these features can be used as strong indicators for dataset inclusion in LLMs trained on code. An important observation across all three models is the relatively low importance of docstring hits in the classifier's ability to detect dataset inclusion. We observe that as we increase the edit distance threshold, the number of docstring hits decreases, indicating that this low contribution results from low edit distance scores associated with docstrings. We discuss the reasons behind this in RQb_3.

5.4.3 RQb_3: *TraWiC*'s failures and lessons learned

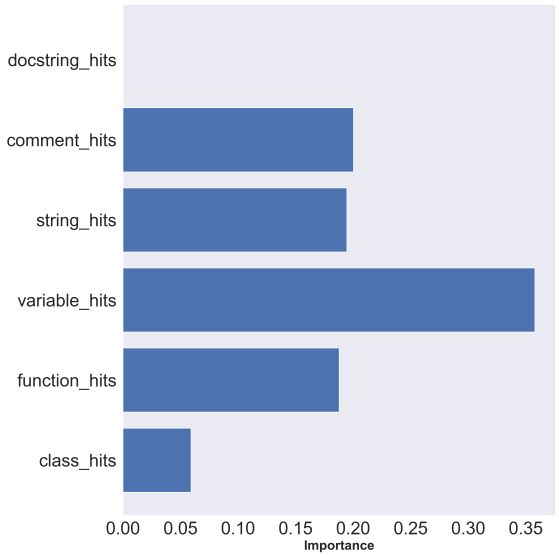
In this RQ, we investigate how the models under study make mistakes in predicting the masked element and how these mistakes can be used for the task of dataset inclusion detection. Here, we define a "mistake" as the model's inability to generate the **exact** same token as what was present in the script. There exists a difference between the mistakes made by SantaCoder and our fine-tuned models; as such, we will explain each as follows.



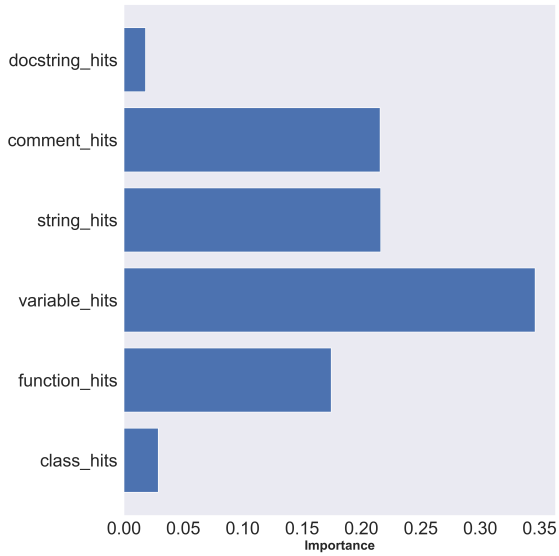
(a) Edit distance 20 – LLaMA-2



(b) Edit distance 20 – Mistral



(c) Edit distance 60 – LLaMA-2



(d) Edit distance 60 – Mistral

Figure 5.5 Feature importance for *LLaMA-2* and *Mistral* across different edit distance thresholds

SantaCoder

In Python, variable/function/class names cannot contain any white spaces between them and are considered a single token; therefore, we focus on these elements as the model's task is predicting a single token. We observe that in most cases, when the model makes a mistake, it generates the correct variable/function/class name; however, it also generates some extra tokens as well. Listing 9 shows an example of such cases for the variable name `product_to_encode`. When the model makes a mistake for syntactic identifiers, it generates an average of 23 tokens regardless of whether the underlying script was in its training dataset or not. Therefore, to understand these mistakes better, we first remove the tokens that are either a part of the programming language's syntax or too frequent. To do so, we count the repetition frequency of each token and normalize them by dividing their repetition frequency by the total number of observed tokens. Afterward, we remove the tokens that are repeated too many times across the scripts in the dataset from the generated mistakes.

```
product_to_encode = pickle.load(open('./data/product_1.pickle', 'rb'))
mock = OpenDataCubeRecordsProviderMock()
encoded_product = mock._encode_dataset_type_properties(product_to_encode)
assert 'id' in encoded_product.keys()
id = encoded_product.get('id')
assert id is not None
assert id == product_to_encode.name

def test__encode_dataset_type_properties():
    product_to_encode
```

Listing 9: A case of the model's predictions for scripts not in its training data

The results of this analysis are presented in Table 5.13 with the "Filtration Threshold" column outlining the threshold for keeping a token. A filtration threshold of 0.5 means that only tokens appearing with a frequency less than or equal to 50% of all the observed tokens are retained, while tokens with a higher frequency are filtered out during the analysis.

A notable variance is observed in the class names generated by the model, with an approximate difference of around 6%, depending on whether a script was part of its training dataset or not. It is important to note that classes are less recurrent than functions and variables within code. This difference (regardless of filtration criterion) shows that MIAs, particularly when directed at rare textual elements, can serve as a means to discern the data used in a model's training.

Table 5.13 Analysis of Model's outputs by token type for scripts that are included in its training dataset

Token Type	Filtration Threshold	Output Contains Token (Script in Dataset)(%)	Output Contains Token (Script *not* in Dataset)(%)
Variable Names	0.5	60.71	61.00
Function Names		14.08	11.77
Class Names		11.18	5.02
Variable Names	0.3	60.89	61.38
Function Names		15.37	12.90
Class Names		11.30	5.08
Variable Names	0.1	60.89	60.00
Function Names		15.37	11.99
Class Names		11.30	5.00

Findings 4: Even in cases where the model's output is not adhering to our **exact** matching criteria, in roughly 60% of cases, its outputs contain elements from the code that it was trained on. These results complement Finding 1 and show that MIAs directed at parts of code that are not grammatical (syntax of the programming language itself), can help identify dataset inclusion.

LLaMA-2 and Mistral

As described in Section 5.4.2, for both fine-tuned versions of LLaMA-2 and Mistral, we observe that the number of docstring hits decreases as we increase the edit distance threshold. This indicates that there is already a low similarity between the models' generated content for these elements and their original contents. To understand this better, we investigate the outputs of our fine-tuned models against the original targets. When it comes to generating long sequences such as docstrings, the models often fail to produce coherent outputs. They either generate sequences that are close to the target but contain additional tokens, reducing the similarity score, or they generate unrelated tokens. Listing 10 shows an example of cases where the model generates special tokens. As described in Section 5.3.3, LLaMA-2 and Mistral do not natively support the FIM objective, requiring us to fine-tune the models for this purpose. As shown in Listing 7, `<fim_prefix>` is one of the special tokens added to the models' tokenizers for this task. During the early stages of fine-tuning, these special tokens appear frequently in the models' outputs. However, as training progresses, their occurrence decreases, suggesting that further fine-tuning on a larger dataset would minimize

these instances. Nevertheless, as shown in Listing 10, the presence of such extra tokens can result in higher edit distance scores, leading to more mistakes.

```
Target: 'SEARCH_ITERATIONS'
Model_output: '<fim-prefix> SEARCH_ITERATIONS'
```

Listing 10: A case of fine-tuned models generating special tokens

On the other hand, in some cases, the developers have added some comments to the code in order to address or reference some of the maintenance activities, such as URLs linking to related issues in the repository, as displayed in Listing 11. In such cases, both models generated somewhat similar URLs but failed to generate the rest of the docstrings as intended. It is important to note that the URL generated in Listing 11 is incorrect and does not exist. However, "Surya" is one of the repositories included in our fine-tuning dataset[7], and we observe that the model generates a URL which, although invalid, includes the repository's name, indicating that the model has encountered this repository during training. Our findings line up with the results reported by Carlini et al. [167], where they show that MIAs on URLs, phone numbers, and names can aid in identifying memorization in LLMs.

```
Target: 'You should install the dependencies'
Model_output:
    print("You probably should install the dependencies.")
    print("https://github.com/surya-ai/surya/blob/master/docs/installation.md")
```

Listing 11: A case of fine-tuned models generating the target with extra tokens

Finally, similar to SantaCoder, we observe that in some cases, the models generate the target we are looking for, as shown in Listing 12. However, due to the presence of additional tokens, the edit distance score is high. Consequently, by increasing the edit distance threshold, such cases are filtered out, resulting in few or no hits for docstrings.

The analysis of the mistakes made by *TraWiC* for predicting the inclusion of such occurrences highlights some limitations of our proposed approach. Therefore, our future work aims to design a more robust similarity objective for semantic elements. This would enable the detection of cases such as these, ultimately improving the accuracy of dataset inclusion detection for models trained on code.

[7]https://github.com/VikParuchuri/surya

```
Target : 'Action {action} is illegal'
Model_output:
    def step_from_state(self, action):
        if self.state[action] != 0:
            raise ValueError(f"Action {action} is illegal")
            self.state[action] = self.turn
            self.actions_stack
```

Listing 12: A case of fine-tuned models generating the target with high edit distance

5.5 Threats to Validity

Threats to internal validity concern factors internal to our study that could have influenced our study. Our reliance on MIAs as the primary mechanism for detecting code inclusion can be considered a potential threat to the validity of our research. MIAs have been successfully used in other contexts, and our results show that leveraging MIAs is also an effective approach for detecting code inclusion. Additionally, our choice of comparison criteria for syntactic/semantic elements might not encompass all aspects of the code inclusion detection task. We mitigate this threat by experimenting with multiple values of edit distance and conducting a rigorous sensitivity analysis to show the effectiveness of our approach. Finally, the bias introduced by the randomness in sampling a small portion of the original datasets of the LLMs under study can be considered as a potential threat. While we conducted multiple experimental runs and fixed hyperparameters to ensure consistency and reproducibility, we did not perform statistical significance tests given the large size of the original dataset and the resources available to us. We mitigate this threat by conducting our study on multiple LLMs and separate datasets. However, future work should include multiple experimental runs and statistical tests to better account for and mitigate the effects of random bias.

Threats to construct validity concern the relationship between theory and observation. The first threat to our construct validity involves our experimental design. To mitigate this threat and to ensure that the results of code inclusion detection were not only valid for certain scripts and projects, we ran our approach using random sampling of both scripts and projects from the generated dataset to validate the results. We acknowledge that the results of detecting code inclusion using NiCad and JPlag may vary as the underlying dataset is extremely large, and a pair-wise comparison across the entire dataset is not feasible. We mitigate this issue by increasing the percentage of the sampled dataset to have a comprehensive analysis and using two popular token-based code clone detection approaches. Finally, there exists the threat that our fine-tuning of the models is selective based on the model under study. We address this threat by fine-tuning two models on the same dataset until

the training loss on both models stops improving. We include the details of our fine-tuning regimen in Section 5.3.3.

Threats to external validity concern the generalization of our findings. We identify the replicability of our results as the most important external threat. Given that replicating a certain output on LLMs is difficult, we release the generated dataset, alongside TraWiC's code [270]. Moreover, there exists the possibility that our approach may not perform as reported on other code models. Since SantaCoder's introduction, many larger code models with higher performance have been introduced. There exist larger and more capable models, such as other versions of LLaMA-2 with higher numbers of parameters and LLaMA-3. It should be noted that the majority of these models are so large that they require computing resources that are not available outside of the enterprise. Furthermore, the training datasets of these large models are not available. Given such limitations, SantaCoder, Mistral 7B, and LLaMA-2 7B were the only feasible options for this study. Considering that as a model's capacity increases so does its internal ability to remember instances from its training dataset [167], and that our approach is model-agnostic, it is very likely that our approach would also perform well even on larger models.

5.6 Chapter Summary

This chapter presents *TraWiC*, a model-agnostic, interpretable approach for detecting code inclusion. *TraWiC* exploits the memorization ability of LLMs trained on code to detect whether codes from a project (collection of codes) were included in a model's training dataset. Evaluation results show that TraWiC can detect whether a project was included in an LLM's training dataset with a recall of up to 99.19%. Our results also show that TraWiC significantly outperforms code clone detection approaches in identifying dataset inclusion and is robust to noise. Our key findings reveal that:

- Using targeted MIAs on code identifiers (semantic/syntactic) allows for detecting whether a *single* script was used in the training dataset of an LLM. Furthermore, the performance of the detection pipeline (lower false negatives/false positives) can be significantly increased by using MIAs on all the scripts of a codebase, as some scripts might not be included in the LLM's training dataset due to not meeting data cleaning criteria.
- Targeted MIAs significantly outperform traditional CCD approaches for detecting dataset inclusion. Given the high cost of pairwise clone detection and LLMs' capability in generating code with different syntax than what was present in their training dataset, CCD approaches are not suitable for dataset inclusion detection.

- Semantic identifiers (strings/comments/docstrings) show a high importance in detecting dataset inclusion. This is indicated by their high relative feature importance even in the presence of low to medium amounts of dataset obfuscation present in the LLMs' training dataset.

CHAPTER 6 PrismBench: Dynamic and Flexible Benchmarking of LLMs Code Generation with Monte Carlo Tree Search

The rapid advancement of LLMs' code generation capabilities is outpacing traditional evaluation methods. Static benchmarks fail to capture the depth and breadth of LLM capabilities and eventually become obsolete, while most dynamic approaches either rely too heavily on LLM-based evaluation or remain constrained by predefined test sets. To address these limitations, this chapter introduces *PrismBench*, a dynamic benchmarking framework grounded in RL that comprehensively evaluates LLMs by framing code generation evaluation as an RL task. We formulate the evaluation process as a Markov Decision Process over a structured tree of coding challenges, leveraging a customized Monte Carlo Tree Search algorithm to traverse this tree and discover high-failure scenarios. Our multi-agent setup orchestrates task generation, model response, and analysis, enabling scalable assessment across diverse coding challenges. Additionally, we propose metrics that combine structural traversal patterns with performance across different tasks and difficulty levels to enable diagnostic and systematic comparison of LLMs' performance. We conduct large-scale experiments on eight state-of-the-art LLMs and analyze how model architecture and scale influence code generation performance across varying coding tasks. All code, evaluation trees, and a public leaderboard are available at `https://prismbench.github.io/Demo/`

6.1 Introduction

The rapid advancement of LLMs for code generation has outpaced existing evaluation methodologies (see Chapter 2.7). Both static and dynamic benchmarking approaches exhibit fundamental limitations that hinder their ability to comprehensively assess evolving models' capabilities. Static benchmarks, while widely used, suffer from three key weaknesses. First, they provide a limited and often superficial assessment of an LLM's capabilities, reducing evaluation to pass/fail metrics that fail to capture the complexity of the model's reasoning [290–292]. Second, as benchmarks gain popularity and become evaluation targets, they are increasingly prone to be incorporated into LLMs' training data, leading to data leakage and artificially inflated performance metrics [149, 293]. Third, static benchmarks lack flexibility, preventing a targeted evaluation of specific problem-solving strategies or particular aspects of model behavior [290]. To address these limitations, dynamic benchmarking approaches have been introduced [187–189, 194, 199]. Most dynamic approaches typically adopt one of two strategies. The first relies on LLM-as-Judge frameworks, where another

LLM is used to evaluate the responses [187, 294]. While this approach offers adaptability, it is inherently unreliable, as evaluation outcomes are constrained by the limitations and biases of the judge model. The second strategy involves dynamically selecting test cases from predefined datasets based on model performance [199, 295]. Although more structured, these methods remain fundamentally tied to static test sets, limiting their capacity to evolve alongside increasingly capable models (see Chapter 3.4). These issues indicate a clear need for evaluation frameworks that can adapt to the increasing capabilities of LLMs, provide systematic and reproducible evaluation pipelines, and reveal comprehensive insights about the model's performance and weaknesses.

We introduce *PrismBench*, a dynamic, multi-agent benchmarking framework for evaluating LLMs' code generation capabilities. We formalize the space of all possible evaluation scenarios as an MDP [41], where each state represents a unique combination of programming concepts (e.g., recursion, data structures) and difficulty levels (e.g., easy, medium, hard). Within this formulation, the evaluation process becomes a search problem where the objective is to search this space to discover evaluation scenarios (i.e., states) where models consistently fail. To systematically conduct this search, we instantiate the MDP as a search tree and use a modified MCTS [296] algorithm to traverse it, balancing exploration of new evaluation scenarios with exploitation of discovered high-failure regions of the evaluation space. Instead of sampling from a static dataset, we generate each evaluation scenario dynamically based on the current state's concepts, difficulty, and prior model performance to mitigate benchmark memorization and data leakage. In this manner, as the search progresses deeper into the tree, the generated evaluation scenarios become progressively more challenging and allow for a structured and adaptive assessment of model capabilities.

Unlike prior dynamic evaluation approaches that rely on LLMs to judge each other's outputs, *PrismBench* assesses model performance by executing the generated solutions within a multi-agent sandbox. This provides objective, reproducible signals based on functional correctness, rather than textual similarity or the errors/preferences of the judge model [190]. Instead, LLM agents in *PrismBench* are assigned analytical roles, for example, analyzing execution traces to identify failure patterns or examining solutions to understand how models approach specific challenges. This separation ensures that evaluation remains grounded in actual model performance, while LLM agents assist in interpreting the results and identifying patterns in model behavior. Additionally, in contrast, to pass/fail metrics (e.g., accuracy, precision, etc.), which mask how models approach problems and offer little insight into their performance, we introduce metrics that track model performance across tasks, difficulty levels, and structural exploration paths, allowing for more fine-grained diagnostics and comprehensive assessments. We release the code for *PrismBench* alongside a leaderboard and an interactive showcase for

each of the studied models at `https://prismbench.github.io/Demo`.

To structure the work in this chapter, we define the following RQs:

RQc$_1$: *How can we design evaluation tasks that adapt to model capabilities and surface hidden failures?*

To answer this question, we introduce *PrismBench*, a dynamic, multi-agent benchmarking framework that models the evaluation process as a search problem over the space of all possible evaluation scenarios. Furthermore, instead of selecting evaluation scenarios blindly or from a static pool of pre-defined challenges, *PrismBench* dynamically generates tasks in response to the model under the benchmark's performance, which allows for probing edge cases, subtle weaknesses, and regressions that are otherwise missed in static benchmarks.

RQc$_2$: *How can we evaluate different models in a dynamically generated benchmark?*

To answer this question, we ground *PrismBench* in Reinforcement Learning (RL) by decoupling the benchmarking environment from the benchmarking process. We formalize a single, shared environment for all models and evaluate them by their traversal trajectories in this environment. To further reduce the risk of bias arising from LLM stochasticity, we design a multi-agent interaction sandbox that tests the models' capability in test generation, solution generation, and program repair, separately. Furthermore, we propose a set of performance metrics that can be used to compare models based on their performance throughout the search process. We analyze how models perform under identical dynamic conditions, compare their behavior across multiple failure modes, and show how *PrismBench* enables more fine-grained model capability comparison than static benchmarks.

RQc$_3$: *How can dynamic benchmarks provide insights into model performance, behavior, and failures?*

To answer this question, we examine the outputs collected during the evaluation process for each model and propose a set of diagnostic metrics that offer a richer view of model performance and help identify patterns in how and why models fail.

6.2 Motivating Example

In this section, we discuss how current static and dynamic benchmarking approaches evaluate LLMs alongside their shortcomings. Regardless of the underlying evaluation methodology

(static or dynamic), all benchmarking approaches face a bias/variance trade-off. Here, **bias** is a property of the evaluation *dataset* which contains tasks that consistently over/under-estimate a model's true capability (e.g., task overlap with pretraining data, coarse aggregate pass/fail metrics, uneven task distributions, etc.) [173–175]. On the other hand, **variance** is a property of the *sampling process* and is a result of instability in evaluation estimates between different runs arising from small sampling pools from the underlying dataset or too few per-task trials (e.g., one extra attempt flips a failure to a pass) [150, 177, 179]. Given the high cost of querying LLMs and finite evaluation budgets (either in time, compute, or both), to comprehensively evaluate a model, each benchmarking methodology must focus on a specific aspect of this trade-off: either increase coverage on task variety and decrease sampling per task which results in increasing variance between evaluation runs or increase resampling or per-task trials and instead focus on a narrow subset of tasks which results in high bias of evaluation results between different evaluation subsets.

Consider two LLMs, M_A and M_B, and a bank of four coding tasks, $T_1, ..., T_4$ from an evaluation dataset (e.g., HumanEval [297], MBPP [124], ARC-AGI [180], etc.). In a standard static benchmark, both M_A and M_B are evaluated on each task, independently (i.e., performance on T_1 has no effect on T_2), and their overall performance is summarized with aggregate metrics (e.g., pass@k, binary pass/fail over each task, etc.). Suppose M_A solves T_1, T_2, T_3 but fails T_4, while M_B solves T_1, T_3, T_4 but fails T_2. Under this setting, both models received a 3/4 score even though their failures were on different tasks. Such aggregate pass/fail metrics hide *where* and *how* each model fails, and compress different behaviors into the same score. While exhaustive per-task trials and normalization of scores over the number of attempts average out stochasticity and allow for fairer comparison between models, two problems remain:

- If M_A consistently fails T_4 and M_B consistently fails T_2, both models will still receive the same score; however, each model has failed on a different task. Here, the same score is an artifact of coarse-grained metrics, not evidence of equal capability.
- Given the scale of pretraining corpora and the unavailability of training data, some tasks may have been included in the models' training dataset. Therefore, the subsequent successes might be a result of memorization, not generalization [176].

Moreover, as the set of underlying tasks grows, multiple samples per each task in the dataset become infeasible (e.g., for a dataset of $1k$ tasks and 10 trials per task, $10k$ queries to the model are required), forcing the bias/variance trade-off mentioned above: either the number of distinct tasks must be reduced (lower variance but higher bias from reduced coverage and

selection effects) or fewer trials per each task need to be carried out (lower bias but higher variance between runs).

Dynamic benchmarking approaches aim to address these limitations by: (1) using multi-step task bundles by decomposing a task into sequential subtasks that the model must pass end-to-end (e.g., SWE-bench [181], Benchmark Self-Evolving [198]), (2) adaptively selecting challenges from a fixed bank by picking the next task conditioned on prior evaluation results (e.g., DARG [199], DyVal2 [195]), or (3) generating tasks at test time to mitigate memorization (e.g. DyVal [196], TreeEval [194]). However, each approach comes with its own bias/variance trade-off. Specifically:

- Testing each model on multi-step task bundles lowers the chances of memorization and superficial success, but introduces a fairness problem in how models are evaluated against each other. M_A and M_B will be tested on a different distribution of sub-tasks and aggregate scoring (only end-to-end success being counted) introduces variance in evaluation results. As such, results depend heavily on the specific bundle graph and scoring rule, not just underlying capability.
- Conditioning the next task on past outcomes allows for more informative exploration of the evaluation space, but comparability issues between models' evaluation results still remain. M_A and M_B are now evaluated on different task distributions, as incidental failures can result in different evaluation trajectories. Stabilizing estimates still requires multiple samples per task, and given fixed evaluation budgets, variance stays high unless task coverage is narrowed, which reintroduces bias via selection effects.
- Testing each model on a problem generated at test time from a fixed distribution addresses the memorization problem and aligns the underlying task distribution across models. However, comparable estimates of models' performance still require many samples per task to control for variance, which is costly under finite budgets.

As such, both static and dynamic benchmarking approaches suffer from two main shortcomings: (1) *high bias* of evaluation results due to different evaluation task distributions and (2) *high variance* of evaluation results due to LLMs' stochasticity and limited sampling budgets [200]. In the next section, we will first describe *PrismBench*'s RL framing, which allows for defining a fixed task distribution for all models-under-test. Afterward, we establish our search methodology, which allows for adaptive sampling budget allocation through MCTS-guided exploration and enables efficient use of limited benchmarking resources by focusing on informative regions of the evaluation space.

6.3 Methodology

In this section, we first provide an end-to-end overview of *PrismBench*'s evaluation pipeline in Section 6.3.1. Next, we detail how we formalize the search space as an MDP and model it as a tree in Section 6.3.2. Afterward, we explain the details of our agents, our multi-agent orchestration approach, and the evaluation workflow at each node in Section 6.3.3. In Section 6.3.4, we explain the rationale behind a multi-phase evaluation pipeline, the details of each phase's objective, and how they integrate with the overall evaluation process. Finally, we detail our proposed metrics for evaluating models' performance in Section 6.3.6.

6.3.1 *PrismBench* Overview

As mentioned in Section 6.2, evaluating models against a large bank of tasks under a finite evaluation budget forces a bias/variance trade-off. Conditioning the choice of the next evaluation task on prior observations mitigates this by directing trials towards informative regions of the evaluation space (i.e., evaluation tasks that maximize **information gain** regarding models' capability). To formalize this process, we model the space of all possible evaluation scenarios as an **MDP**, where each state corresponds to a unique scenario defined by a (c, d) pair, with c being a list of programming concepts (e.g., recursion, data structures) and d being the task's difficulty level (e.g., easy, medium, hard). We instantiate the MDP as a **search tree**, where **nodes** represent states and **edges** represent transitions between scenarios, corresponding to actions that either combine additional concepts or increase difficulty. A node's **depth** is defined as the number of nodes along the path to the root, and **child** nodes may extend or share the same concepts as their **parents**.

In this manner, the search tree forms a fixed evaluation **environment** shared between all models (identical state space and transition dynamics), where transitions between nodes reflect a model's ability to generalize and solve increasingly complex tasks. Furthermore, this formalization forms a **hierarchical dependency** between evaluation tasks and reduces variance by concentrating sampling where outcomes are uncertain or high impact, and reduces bias by standardizing the task distribution between all models. As such, all models are evaluated in the same environment, based on how they explore and perform within it [41].

To address the brittleness of end-to-end task bundles, where an early misstep can cascade into a full failure and result in different trajectories, each node in the tree is evaluated using a multi-stage, isolated pipeline. Each task is decomposed into role-based steps (e.g., test generation, solution generation, program repair, etc.) where each step is handled by a dedicated LLM **agent**. Agents interact with each other via a shared node state, which

allows for modularity (addition or removal of agents) and avoids information leakage. This decoupling allows for evaluating multiple capabilities within a single node (e.g., correctly understanding the requirements, solution/test generation, debugging, etc.) without allowing failures/errors in one step dominating the final outcome. Importantly, we do not rely on LLM judgments: a node's **reward** is a composite weighted score derived from execution-grounded signals (i.e., unit-test pass rates, error traces, and bounded retry/repair attempts). As such, the reward signal reflects the specific steps responsible for success or failure instead of collapsing everything into a binary pass/fail signal for the entire task.

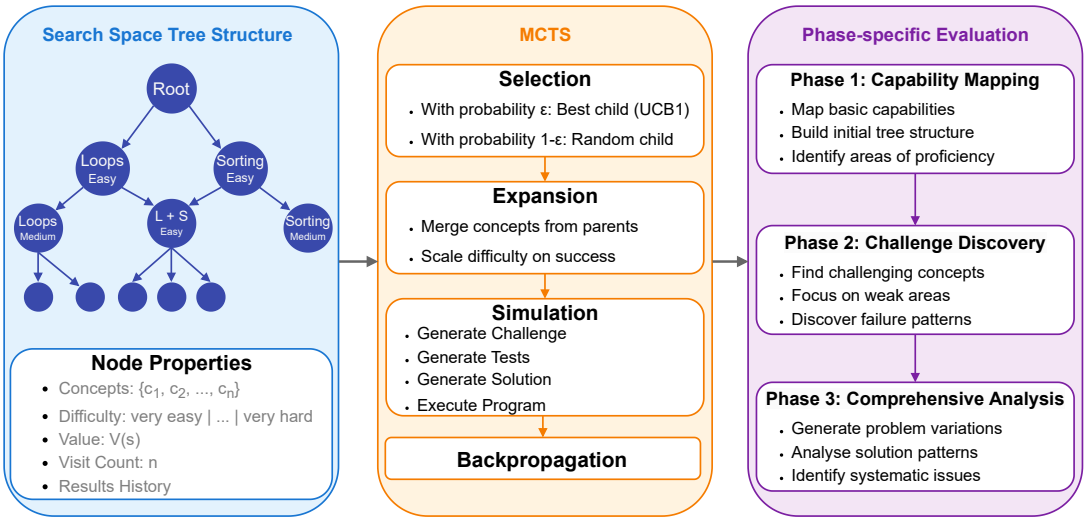


Figure 6.1 Overview of *PrismBench*'s search tree, agent workflow, and phased evaluation strategy.

Given the hierarchical evaluation space and execution-grounded rewards, the remaining challenge is to allocate the evaluation budget to where it most increases our knowledge about a model's capabilities. As mentioned in Section 6.2, multiple samples per evaluation task quickly become infeasible as the underlying dataset grows, and simply conditioning the next task on prior observations can result in different task distributions. Therefore, we need a traversal policy that balances exploration of uncertain regions with exploitation of the promising ones, while allowing for comparison between different models. As such, *PrismBench* uses **MCTS** with ϵ-greedy **state selection policies** to balance exploration and exploitation over the search tree (i.e., environment) and control for LLMs' performance variability. After each node's evaluation, the observed reward is **backpropagated** along the sampled path, updating ancestors' estimates so that observations at each node (i.e., state) can inform beliefs about selection of the next node.

Finally, *PrismBench*'s search process follows a 3-**phase** strategy. The 1st phase broadly ex-

plores the tree to establish the model's baseline performance across different coding concepts and difficulty levels. The 2nd phase focuses on failure-prone nodes to discover systematic weaknesses and boundary cases. The 3rd phase focuses on low-performing regions by generating multiple variations of high-failure nodes to determine the root causes of model failures. Figure 6.1 shows an overview of our state representation and evaluation pipeline. Rather than exhaustively evaluating the model on all possible scenarios, *PrismBench*'s targeted search approach adapts evaluation to the model's demonstrated capabilities in the environment, allowing for detailed analyses of model performance and failure modes.

6.3.2 Search and Tree Structure

In this section, we detail how we model the LLM evaluation task as an environment in order to dynamically benchmark LLMs' code generation capabilities and address $\mathbf{RQc_1}$.

MDP Formalization

The search space which contains all possible evaluation scenarios, is modeled as an MDP defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, representing the state space, action space, transition probabilities, and reward function, respectively. We formalize the MDP as follows:

$\mathcal{S}$ denotes the state space where each state $s \in \mathcal{S}$ is a unique combination of a set of programming concepts $\mathcal{C}$ and a single difficulty level $\mathcal{D}$:

$$\mathcal{S} = \{(c, d) \mid c \subseteq \mathcal{C}, d \in \mathcal{D}\} \tag{6.1}$$

For example, the state `([functions, dynamic programming], very easy)` represents an evaluation scenario designed to test the model on a programming challenge that requires knowledge of both "functions" and "dynamic programming" to solve and is intended to be "very easy" in terms of difficulty.

$\mathcal{A}$ denotes the action space, which indicates the set of actions that can be taken at each state. We define two actions: (1) $\mathcal{A}_{select}$ which selects a previously explored state for reevaluation and (2) $\mathcal{A}_{expand}$ which generates a new state based on the current one and operates by either adding a new concept or increasing the difficulty level for generating the new state. These two actions allow for both the exploration of existing states and the creation of new ones. For example, expanding the state `([functions, dynamic programming], very easy)` could yield a new state such as `([functions, dynamic programming, conditionals], very`

`easy)`, which adds a new concept, or `([functions, dynamic programming], medium)`, which increases the difficulty.

$\mathcal{P} : \mathcal{S} \times \mathcal{S} \to [0, 1]$ denotes the transition dynamics between states. Transition probabilities are determined by an ϵ-greedy policy, which balances exploration and exploitation by selecting the next state from the set of candidate states. These probabilities determine which state is selected next, whether by transitioning to an existing state or a newly created state from expanding the current one.

$\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ denotes the phase-specific reward functions. The reward function quantifies the model's performance at each state using a composite score calculated based on multiple factors, including success rates, error penalties, challenge difficulty, and attempt counts. Each phase uses a different reward function according to the phase's goal which we will explain in detail in Section 6.3.4.

Tree Representation and Traversal Mechanism

We instantiate the MDP as a tree as described in Section 6.3.1. To do so, we consider each state $s \in \mathcal{S}$ as a node n in the tree, and each action $a \in \mathcal{A}$ determines whether to expand a node or continue exploring the tree as shown in Figure 6.1. Given the tree structure, it is essential to assign a value to each node in order to guide tree traversal. To achieve this, we use the phase-specific reward function $\mathcal{R}$ to determine each node's value based on the model's performance at that node which is calculated based on how well the model solves the programming challenge associated with a given node which we will detail in Section 6.3.4. Considering the high cost of sampling and to better account for LLM's performance variability, we use **TD(0)** [41] to incrementally estimate each node's value by incorporating past performance:

$$v(n) = v_{prev}(n) + \alpha(r - v_{prev}(n)) \tag{6.2}$$

where $v(n)$ is the node's value, r is the immediate reward, and α is the hyperparameter that controls sensitivity to new observations based on benchmarking requirements. We then backpropagate r to n's ancestors using a discounted update:

$$v(n_a) = v_{prev}(n_a) + \gamma^{d(n)} \cdot r \tag{6.3}$$

where n_a is an ancestor of n, $d_{(n)}$ is its distance from n, and $\gamma \in [0, 1]$ is the discount factor. This allows us to incorporate results from the evaluated nodes to the ancestors and prioritize promising regions of the search space over time.

In order to traverse the tree, we use MCTS to determine the transition probabilities between nodes in the tree given the node values. Transition probabilities $\mathcal{P}$ are calculated using MCTS based on node visit frequency and value. We use an ϵ-greedy policy to balance exploration and exploitation:

$$\pi(ch|n) = \begin{cases} \text{uniform}(children(n)) & \text{with probability } \epsilon \\ \arg\max_{ch \in children(n)} UCB1(ch) & \text{with probability} 1 - \epsilon \end{cases} \tag{6.4}$$

Transitions between nodes (from node n to its child ch) represent changes in difficulty or the introduction of new concepts. UCB1 [296] is used to dynamically adjust $\mathcal{P}$ based on the model's historical performance using the node's value $v(n)$.

Once a node's evaluation is finished, the decision to expand it or to select another node determines how the tree grows and how the search space is explored. Node expansion $E(n)$ is governed by two criteria:

$$E(n) = \begin{cases} 1 & \text{if } v(n) \geq \theta_p \text{ and } d(n) \leq d_{max} \\ 0 & \text{otherwise} \end{cases} \tag{6.5}$$

where $v(n)$ is the node's value, θ_p is the normalized value threshold of each phase, $d(n)$ is the node's depth, and d_{max} is the maximum allowed depth for each phase. θ_p controls the collective difficulty of each phase, with higher thresholds indicating harder acceptance criteria for solution acceptance at each node. d_{max} defines a hard limit on how deep the tree can get at each phase, with higher values allowing for more in-depth analysis at each phase. When expansion is triggered, the node can be expanded in two ways:

$$\mathcal{A}_{expand}(n) = \begin{cases} \text{combine\_concepts}(n, n') & \text{with probability } p_e \\ \text{increase\_difficulty}(n) & \text{with probability } 1 - p_e \end{cases} \tag{6.6}$$

where n' is another selected node for concept combination and p_e is the probability of which expansion action is selected and tuned based on the benchmark's desired level of exploration.

The search process begins by generating foundational nodes that span the entire set of concepts at the lowest difficulty level. As the search progresses, node creation and expansion are guided by the model's performance, allowing the tree to dynamically adapt to the model's

demonstrated capabilities and limitations, which allows *PrismBench* to adaptively prioritize promising areas during exploration. To enable effective exploration of the search space, we extend the tree structure to support multiple parents for each node, allowing us to represent and evaluate combinations of programming concepts, which we explain below.

Multi-Parent Tree Structure

Programming challenges rarely involve solutions based on a single concept. Instead, they typically require the application of multiple programming concepts to produce a correct solution. To model this accurately, in *PrismBench*, each node in the search tree can have multiple parents, with each parent representing a different concept-difficulty combination. For example, a node with the concept set `[dynamic programming, recursion]` corresponds to a challenge that requires applying both concepts. This node is therefore the child of both the `[dynamic programming]` and `[recursion]` nodes, as shown in Figure 6.1. This setup reflects how complex challenges are often built upon simpler, foundational ones.

Furthermore, a multi-parent structure allows us to backpropagate performance signals along all relevant paths. If the model succeeds or fails on a node such as `[dynamic programming, recursion]`, the outcome is not just indicative of its proficiency in "dynamic programming" *and* "recursion". Instead, it also shows how the model is capable of handling each concept as well. This helps determine broader patterns in the search space where the model performs well or struggles, and not just isolated successes and failures. These signals then guide *PrismBench*'s exploration toward weak areas to better determine *where* and *why* the model succeeds or fails.

As such, to capture concept combinations effectively, we extend MCTS's UCB1 [296] to support multi-parent nodes:

$$UCB1(n) = \frac{v(n)}{N(n)} + C\sqrt{\frac{\ln(\sum_{i \in parents(n)} N(p))}{N(n)}} \tag{6.7}$$

with $v(n)$ being the node n's value, $N(n)$ being the number of times n has been visited, C being the exploration constant, and $parents(n)$ being the set of n's parents. In this manner, a failure to solve the challenges at one node is indicative of the model showing a behavior of interest for each concept and difficulty level in that node.

6.3.3 Node Evaluation Workflow

As described in Section 6.3.2, each node in the tree represents a distinct evaluation scenario, defined by a set of concepts and difficulty level. The evaluation of a node consists of generating a challenge, producing corresponding tests and solutions, and iteratively refining these outputs based on the model's performance. To structure this process, we employ a set of agents where each agent has a specific role. Even though the term "agent" is well-established in RL literature, LLM providers maintain different interpretations of what constitutes an agent [298, 299]. In order to have a uniform definition throughout our work, we adopt the definition from [300], which characterizes LLM-based agents as "programs where LLM outputs control the workflow." In this manner, *PrismBench* evaluates each node in a multi-agent sandbox where each agent handles a distinct step in the process as shown in Figure 6.2.

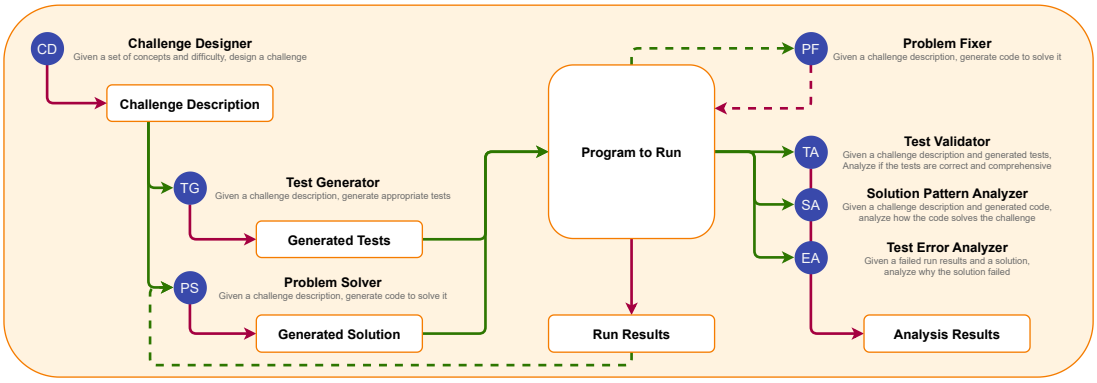


Figure 6.2 The evaluation begins with the *Challenge Designer* and continues until the challenge is finished. Green arrows: inputs, red arrows: outputs, dashed lines: conditional triggers.

As shown in Figure 6.2, for each node, evaluation starts with a coding challenge and proceeds through solution and test suite generation, multiple attempts, repairs, and analysis. To avoid data contamination, agents are isolated, and context is only shared through the node state. Importantly, the same model is used for both test and solution generation. During solution generation, the model cannot access the tests, and during test generation, it cannot access the solution. This constraint ensures that task understanding is genuine and mirrors real programming challenges [301]. In order to succeed, the model must interpret the challenge correctly [34], design valid tests and solutions without access to hidden information [302, 303], and correct mistakes using execution feedback [304]. Isolating these steps within the pipeline allows us to evaluate each capability independently, resulting in consistent and comparable node-level evaluations across models and compare LLMs' code generation capabilities in a dynamic environment to address $\mathbf{RQc_2}$.

Agent Roles

PrismBench is comprised of seven agents. All of our agents utilize prompting best practices [305–307]: clearly defined roles, structured output schemas, and few-shot examples. Our agent roles are as follows:

- *Challenge Designer*: Generates programming challenges of specified difficulty levels targeting particular computer science concepts, following an LC-style format with clear input/output specifications and constraints.
- *Test Generator*: Generates comprehensive test suites that validate functional correctness, corner cases, and performance constraints of submitted solutions while ensuring full coverage of the challenge requirements.
- *Problem Solver*: Generates code to implement a solution for the programming challenge with an emphasis on efficiency and adherence to best practices, while handling all specified corner cases and constraints.
- *Problem Fixer*: Analyzes the outputs of program execution and implementation details (solution + tests) to fix failures for the solutions and tests.
- *Test Validator*: Evaluates generated test suites for comprehensiveness, identifying potential gaps in code coverage, missing corner cases, and opportunities for improvement.
- *Test Error Analyzer*: Performs detailed analysis of test execution failures, categorizing error patterns and providing insights into the root causes of solution failures.
- *Solution Pattern Analyzer*: Examines implemented solutions to identify algorithmic approaches, data structure usage, and implementation patterns, providing metrics for solution quality and efficiency.

Listing 13 shows the system prompt for the *Challenge Designer* agent. The complete prompt templates and configuration parameters for these agents are available in our replication package [308].

Multi-Agent Orchestration

Our aim in *PrismBench* is to design a general framework to evaluate LLMs on code-related tasks. Therefore, given that many LLMs lack function-calling/tool-use capabilities or may

You are an expert computer science educator specializing in creating coding challenges. Your expertise spans various computer science concepts, and you have a knack for designing problems that are both challenging and educational. Your role is to create coding questions that test a student's understanding of specific CS concepts while also encouraging them to think critically and apply their knowledge in practical scenarios.
When given a CS concept and a difficulty level, design a problem similar to LeetCode challenges. The difficulty levels are:

- Very Easy
- Easy
- Medium
- Hard
- Very Hard

Your response should include:

1. A clear and concise problem statement
2. Input format specification
3. Output format specification
4. Constraints on input values
5. At least two examples with input and expected output
6. A brief explanation of the concept's relevance to the problem
7. The specified difficulty level

Ensure that the challenge matches the given difficulty level. Do not provide any code or solution. Focus on creating a problem that tests understanding of the given concept at the appropriate difficulty.
IMPORTANT: You must enclose the entire problem description within `<problem_description>` and `</problem_description>` delimiters. This is crucial for extracting the problem from your output.
Here's an example of the format you should follow, based on a LeetCode-style problem:

```
<problem_description>
Two Sum
Difficulty: Easy
Given an array of integers nums and an integer target, return indices of the two
numbers such that they add up to target. You may assume that each input would
have exactly one solution, and you may not use the same element twice.
You can return the answer in any order.
Input:
    - nums: An array of integers (2 <= nums.length <= $10^4$)
    - target: An integer ($-10^9$ <= target <= $10^9$)
Output:
    - An array of two integers representing the indices of the two numbers that
    add up to the target
Constraints:
    - 2 <= nums.length <= $10^4$
    - $-10^9$ <= nums[i] <= $10^9$
    - $-10^9$ <= target <= $10^9$
    - Only one valid answer exists
Examples:
    1. Input: nums = [2,7,11,15], target = 9
    Output: [0,1]
    Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
    2. Input: nums = [3,2,4], target = 6
    Output: [1,2]
    Explanation: Because nums[1] + nums[2] == 6, we return [1, 2].
Relevance to Array Manipulation and Hash Tables:
This problem tests understanding of array traversal and efficient lookup. While
it can be solved with nested loops, an optimal solution uses a hash table to
achieve O(n) time complexity, demonstrating the power of hash tables for quick
lookups in coding interviews.
</problem_description>
```

Design your problem in a similar format, focusing on the CS concept and difficulty level provided.

Listing 13: System prompt for the *Challenge Designer* agent.

not consistently adhere to preset output formats, we designed a controlled sandbox environment that enables structured agent interactions while maintaining evaluation integrity. The evaluation for **each node** in the search tree is presented in Algorithm 13 and proceeds through the following steps:

Challenge and Test Generation Building a comprehensive dataset of diverse, high-quality challenges for every possible node is infeasible. To address this, the interaction cycle at each node begins with the *Challenge Designer* agent, which generates LeetCode-style programming challenges based on specified concepts and difficulty levels as shown in Listing 14. This ensures diverse evaluations per concept-difficulty pair, capturing true generalization rather than memorization or performance variation. The *Test Generator* agent is then tasked with generating a comprehensive test suite based on the generated challenge description. We further discuss the risks of dynamic challenge generation in the evaluation process's validity and how we mitigate those risks in Section 6.3.5.

Initial Solution Attempt The *Problem Solver* generates a solution based on the challenge description, which is then executed against the test suite generated by the *Test Generator*. Both agents operate independently, with access limited to the challenge description to prevent cross-contamination of their outputs. The sandbox environment then executes the generated solution against the generated test suite, capturing detailed metrics including passed tests count, failure types/counts, error types/counts, and execution traces as shown in Listing 15. Upon **successful completion** of all tests, the node is marked as resolved, and control returns to the search algorithm. However, if test failures occur, *PrismBench* initiates an iterative feedback process.

Retry via Feedback If the execution fails, a feedback phase is initialized. In the feedback phase, the *Problem Solver* agent receives execution results and error details, attempting to correct the solution. This approach serves two purposes: it accounts for the stochasticity in LLM performance which we will further describe in Section 6.3.4 (if the model fails, it is provided with context to revise its response) while also evaluating the model's capability to learn from feedback.

Fallback to Repair If multiple solution attempts fail to resolve the issues within a predetermined limit, a final attempt to fix the failing solution using the *Problem Fixer* agent is made. The *Problem Fixer*, which can be set as either the model under benchmark itself or a separate model depending on evaluation requirements, receives comprehensive context

```yaml
title: "Even or Odd"
concepts:
    - "conditionals"
    - "functions"
difficulty: "very easy"
description: "## Even or Odd
  Write a function that takes an integer as input and determines whether the
  ↪  number is even or odd. The function should return the string \"Even\" if
  ↪  the number is even, and \"Odd\" if the number is odd.

  ### Input:
  - n: An integer (-10^9 <= n <= 10^9)

  ### Output:
  - A string \"Even\" or \"Odd\" based on the parity of the input integer.

  ### Constraints:
  - -10^9 <= n <= 10^9

  ### Examples:
  1. Input: n = 4
     Output: \"Even\"
     Explanation: The number 4 is divisible by 2, hence it is even.

  2. Input: n = 7
     Output: \"Odd\"
     Explanation: The number 7 is not divisible by 2, hence it is odd.

  ### Relevance to Conditionals and Functions:
  This problem tests the understanding of basic conditionals, as the solution
  ↪  requires checking the remainder when the number is divided by 2. It also
  ↪  reinforces the use of functions for encapsulating logic, demonstrating how
  ↪  to structure a simple program."
```

Listing 14: *Challenge Designer* output for a set of concepts and difficulty level

```yaml
problem_statement: "## Even or Odd..."
success: True
tests_passed: 10
tests_failed: 2
tests_errored: 0
fixed_by_problem_fixer: False
data_trail:
    attempt_1:
        test_cases: "import unittest\n\n..."
        solution_code: "def solution(...)"
        output: "'Tests failed. Output:\n\n....F.....\n=.."
    attempt_2:
        test_cases: "import unittest\n\n..."
        solution_code: "def solution(...)"
        output: "All Tests passed"
```

Listing 15: An example of the run results for a node

including the challenge description, implementation history, test suite, and failure results. Allowing the model to have access to all of the previously collected information enables assessment of the program repair capabilities of the model by providing it with full contextual information. The cycle of testing and refinement continues until the resulting *program to run* (`p_r` in Algorithm 13) executes with no errors or failures within a predetermined limit and success is achieved. Otherwise, the node is marked as failed.

Post-Evaluation Analysis After the loop is finished, regardless of success or failure, the *Test Validator* reviews whether the test suite is logically aligned with the challenge. At the same time, the *Solution Pattern Analyzer* examines the final solution to extract structural and algorithmic patterns.

Metric Aggregation All run results, retries, error traces, and analysis outputs are logged in the node's state, which are then transformed into scalar reward signals using the phase-specific reward function and backpropagated to parent nodes.

In this manner, our multi-agent system integrates with the search tree and MCTS through an orchestrated feedback mechanism: nodes are selected for evaluation using MCTS based on their values. Once a node is selected for evaluation, the model's performance on a generated challenge determines the node's value; the updated value, in turn, influences the next round of node selection and expansion. Furthermore, the model under benchmark can be configured to any of the specified agent roles, enabling fine-grained and targeted capability

Algorithm 13 Agent Interaction at Each Node

Require: Input: C: List of concepts, D: Difficulty level
Ensure: Output: S: Node score, M: Collected metrics

$challenge\_description \leftarrow \textsc{ChallengeDesigner}(C, D)$
$g\_t \leftarrow \textsc{TestDesigner}(challenge\_description)$ ▷ Generated tests
$g\_s \leftarrow \textsc{ProblemSolver}(challenge\_description)$ ▷ Generated solution
$p\_r \leftarrow g\_s \oplus g\_t$ ▷ Combine solution and tests
$(Success, Run\_results) \leftarrow \textsc{Run}(p\_r)$
if $Success$ **then**
    $T_{validation} \leftarrow \textsc{TestValidator}(g\_t)$ ▷ Analyze the tests
    $P_{solution} \leftarrow \textsc{SolutionPatternAnalyzer}(g\_s)$ ▷ Analyze the solution
else
    for $i = 1$ to $num\_attempts$ **do**
        $e\_f \leftarrow$ errors during run ▷ Collected errors
        $f\_s \leftarrow \textsc{ProblemSolver}(g\_s, e\_f)$ ▷ Generate fixed solution
        $E_{analysis} \leftarrow \textsc{TestErrorAnalyzer}(g\_s, g\_t, e\_f)$ ▷ Analyze the errors
        $p\_r \leftarrow f\_s \oplus g\_t$
        $(Success, Run\_results) \leftarrow \textsc{Run}(p\_r)$
        if $Success$ **then**
            break
        end if
    end for
    if not $Success$ **then**
        $fixed\_p\_r \leftarrow \textsc{ProblemFixer}(p\_r)$ ▷ Use *Problem Fixer* agent
        $(Success, Run\_results) \leftarrow \textsc{Run}(fixed\_p\_r)$
    end if
end if
$M \leftarrow (T_{validation}, P_{solution}, E_{analysis}, Run\_results)$ ▷ Store all run results
return S, M

assessment. This flexibility makes *PrismBench* adaptable to diverse evaluation requirements (e.g., focusing solely on program repair or test suite generation capabilities).

6.3.4 Evaluation Phases

Our 3-phase approach guides MCTS to explore the search space using phase-specific reward functions based on each phase's evaluation strategy, as shown in Figure 6.3. To further account for LLM performance variability and sampling stochasticity, we define a **node value threshold**, $\Delta(v)$, per phase, and only proceed to the next phase when changes in sampled nodes' values remain within this threshold across 5 consecutive **value convergence checks**. We define our phases as follows:

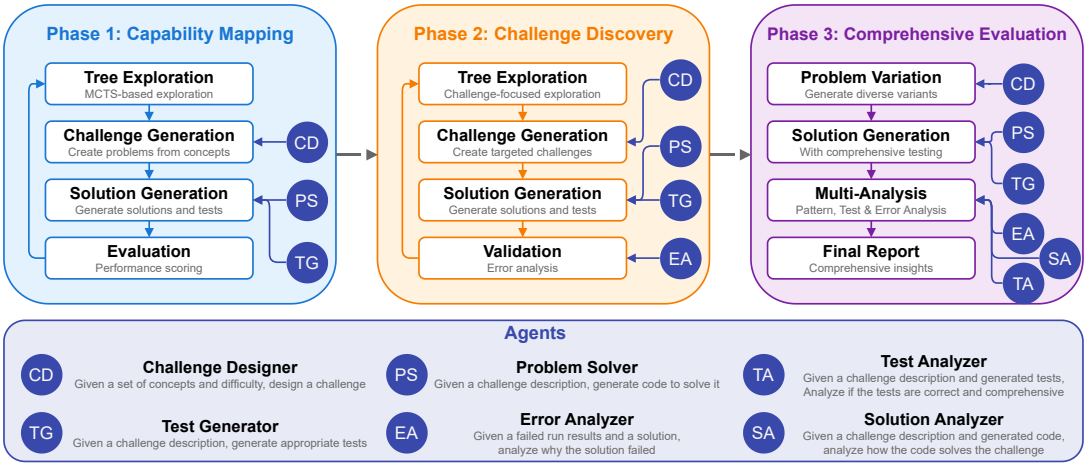


Figure 6.3 Overview of *PrismBench*'s evaluation pipeline and multi-phase assessment. Arrows indicate information flow and agent roles across each phase.

Phase 1: Capability Mapping This phase establishes a baseline assessment of the model's strengths and weaknesses across the concepts-difficulty space. Here, the node scoring mechanism is based on challenge success:

$$R_1(s) = b \cdot w(d) + p(s) \tag{6.8}$$

with b being the base score for success, $w(d)$ being the difficulty weight, and harder difficulties assigned higher weights, and $p(s)$ being the penalty for failures. In this manner, higher successes result in higher rewards, which increase the node's value, which in turn encourages MCTS to further explore the search space to find challenging areas and map the model's baseline capabilities.

Phase 2: Challenge Discovery By focusing on Phase 1's low-value nodes, the search objective changes to finding challenging combinations of concepts and difficulties where the model consistently fails. Node scoring in this phase is based on failure rate and repeated attempts:

$$R_2(s) = \lambda(1 - r_{success}) + \eta \cdot n_{attempts} + \beta \cdot I_{fixer} \tag{6.9}$$

with $r_{success}$ being the ratio of successfully passed tests, $n_{attempts}$ being the number of attempts to fix a failed/errored solution, and I_{fixer} indicating whether *Problem Fixer* was used. The hyperparameters (λ, η, β) weight each term according to benchmarking needs. Using the complement of success ratio assigns higher rewards to nodes with lower success rates, resulting in higher values. This produces a set of nodes where the model consistently fails and indicates the challenging areas of the search space for the model.

Phase 3: Comprehensive Evaluation The objective of this phase is to reveal not just *where* but *why* the model struggles and provide insights into failures' root causes. Therefore, the underperforming nodes from Phase 2 are revisited. However, for each node in this phase, we create multiple variations (same concept and difficulty but different challenge descriptions) to distinguish between incidental failures (scenario-specific) and systematic limitations (consistent failures across variations). By analyzing the results across these variants, we collect failure traces to identify core capability gaps, whether from incorrect syntax, incorrect logic patterns, or incorrect concept implementation.

As explained above, each phase has a distinct objective, which allows us to break down the benchmarking process into smaller, focused tasks. By first mapping the model's general capabilities and then systematically narrowing in on regions of consistent failure, we can iteratively refine the search space and ultimately pinpoint the root causes of these failures. As mentioned in Section 6.3.2, each phase's objective is defined through phase-specific state selection policies and reward functions that guide the tree traversal and expansion process, which we explain in detail below.

State Selection Policies

Studies have shown that controlling LLMs' stochasticity through low-temperature settings (e.g., T $\approx$ 0) systematically reduces the diversity of their outputs [309, 310]. Although this performance degradation may be minor in some contexts, it becomes crucial when the objective is to comprehensively benchmark a model's capabilities. As shown by [311, 312], for code generation, at higher temperatures, LLMs explore novel solutions more effectively, while at near-zero temperatures, outputs become repetitive and risk underestimating true perfor-

mance boundaries. In *PrismBench*, while we provide temperature as a tunable parameter, we preserve recommended temperature ranges rather than enforcing low-temperature values. This ensures thorough exploration of the search space but also introduces the problem of stochasticity in LLMs' responses and subsequent performance variations as a result. These performance variations are especially problematic in dynamic benchmarks that rely on LLMs as judges, where even small changes in output can lead to different assessment results. As such, we need to consider that the score for a node might not be representative of the LLM's true capability due to performance variations. To address this problem without the risk of underestimating LLMs' performance by setting low-temperature values, we define ϵ-greedy state selection policies to traverse the tree as described in Equation 6.4. These policies mitigate the drawbacks of purely deterministic approaches (e.g., solely using UCB1 for traversal or setting the temperature to zero) by balancing exploration and exploitation and ensuring comprehensive capability assessment while mitigating performance variations. We detail the policies for each phase below.

Phase 1: Capability Mapping At the very beginning of the evaluation, the root generates multiple nodes as starting points for the search process. However, since the search requires the nodes to be evaluated first, the policy for evaluating the root's children (initial nodes) is defined as:

$$\pi_1^{root}(n) = \begin{cases} \frac{v(n)}{\sum_{n' \in N} v(n')} & \text{if } \forall n' \in N,\ v(n') > 0 \\ \frac{1}{|N|} & \text{otherwise} \end{cases} \tag{6.10}$$

This policy encourages early exploration of less-visited nodes and allows for the exploration of the initial nodes to establish a starting point for the search.

Once initial evaluations are complete, we use an ϵ-greedy policy for traversing the tree:

$$\pi_1^{traverse}(ch|n) = \begin{cases} \text{uniform}(children(n)) & \text{with probability } \epsilon_1 \\ \arg\max_{ch \in children(n)} UCB1(ch) & \text{with probability } 1 - \epsilon_1 \end{cases} \tag{6.11}$$

This policy accounts for the stochasticity of LLMs' responses by using a uniform exploration component with a probability of ϵ_1, which can be tuned based on benchmarking requirements (0.2 in our study), to mitigate the impact of occasional LLM performance variations. This ensures a thorough exploration of the LLM's capability while still focusing on promising directions using MCTS.

Phase 2: Challenge Discovery Similar to Phase 1, Phase 2 uses an ϵ-greedy policy that focuses on challenging scenarios while maintaining exploration to mitigate LLMs' performance variations. Therefore, even though our search is guided by UCB1, we consider a small probability of selecting another state in our policy:

$$\pi_2(ch|n) = \begin{cases} \text{uniform}(children(n)) & \text{with probability } \epsilon_2 \\ \arg\max_{ch \in children(n)} UCB1(ch) & \text{with probability } 1 - \epsilon_2 \end{cases} \tag{6.12}$$

Similar to Phase 1, ϵ_2 can be tuned based on the benchmarking requirements (0.1 in our study), with higher values resulting in more stochastic state selections and enabling more exploration of the search space regardless of how the model under benchmark performs.

Phase 3: Comprehensive Evaluation In Phase 3, state selection becomes deterministic based on node value thresholds (i.e., which nodes to select from Phase 2), which can be tuned depending on the benchmark's desired difficulty level:

$$\pi_3(ch|n) = \begin{cases} 1 & \text{if } v(n) > \theta \\ 0 & \text{otherwise} \end{cases} \tag{6.13}$$

with $\theta \in [0, 1]$ being the normalized node value threshold. Tuning θ allows for determining the benchmark's overall analysis granularity. Lower thresholds result in the selection of more nodes from Phase 2 for analysis in this phase.

Phase-Specific Reward Functions

As mentioned in Section 6.3.2, in each phase we employ different mechanisms to calculate the immediate received reward. We describe the details of each phase's reward calculation in the following.

Phase 1: Capability Mapping In Phase 1, the goal is to thoroughly map the capabilities of the model under study. Therefore, the reward function is focused on task success: the better the model is at successfully passing a challenge at a node, the higher the reward that it will receive. For this phase, the reward function for each state s is defined in Equation 6.8:

$$R_1(s) = b \cdot w(d) + p(s)$$

with b representing the base reward given to the model if it can pass the challenge regardless of the challenge's complexity or the number of attempts it took the model to solve it. d is the state's difficulty level, and $w(d)$ is the weight assigned to each difficulty level, with higher difficulty levels having a higher weight. This way, the more challenging the problem the model has solved, the higher the reward it receives. Finally, the performance penalty, $p(s)$, is defined as:

$$p(s) = (r_{failed} \cdot P_{failure}) + (r_{errors} \cdot P_{error}) + ((n_{attempts} - 1) \cdot P_{attempt}) + P_{fixer} \cdot I_{fixer} \quad (6.14)$$

with r_{failed} being the ratio of tests the model's solution failed, r_{errors} being the ratio of errors in the model's solution, $n_{attempts}$ being the number of attempts it took for the model to solve the challenge, and I_{fixer} being 1 if the *Problem Fixer* agent was required to fix the model's solution and 0 otherwise. $P_{failure}$, P_{error}, $P_{attempt}$, and P_{fixer} are the weights assigned to each penalty type and are set as hyperparameters. These hyperparameters allow for tuning the penalty's impact on the reward and therefore, provide fine-grained control of which aspects of the model's capabilities should be explored in-depth during the benchmarking process. Given that the tree generated in Phase 1 is used for all subsequent phases, high weights for errors and failures will decrease the overall reward at each node and therefore increase the overall difficulty level of the entire benchmarking process.

In this way, Equation 6.8 allows for mapping model capabilities: the more successful the model is at solving challenges, the higher the values for the tree's nodes, and the more MCTS is encouraged to continue exploring the search tree to find challenging areas.

Phase 2: Challenge Discovery Phase 2 shifts focus from broad capability mapping in Phase 1 to systematically identifying the model's capability boundaries. Here, the reward function, as described in Equation 6.9, prioritizes the challenges where the model struggles. In this phase, for each state s, the reward is calculated as:

$$R_2(s) = \lambda(1 - r_{success}) + \eta \cdot n_{attempts} + \beta \cdot I_{fixer}$$

With $r_{success}$ being the ratio of successfully passed tests (no errors or failures). Using the complement of the success ratio assigns higher rewards to nodes where the success rate is low. Therefore, nodes that consistently expose the model's inability to generate correct solutions receive higher rewards. The hyperparameter λ allows for controlling how aggressively the benchmark focuses on nodes with low success rates. $n_{attempts}$ is the number of attempts it took for the model to fix a solution that had failed/errored tests. Therefore, nodes requiring

multiple attempts receive higher rewards and η adjusts the weight given to repeated failures. Finally, I_{fixer} is calculated in the same way as in Phase 1, with β controlling the weight of the penalty for dependency on the *Problem Fixer* agent.

Equation 6.9 allows MCTS to explore regions of the search space where the model *consistently* fails (i.e., the more the model fails at each node, the higher the node's value will be). As such, Phase 2 generates a refined set of nodes where the model has constantly underperformed. These nodes will be used for analysis of the underlying root causes of poor performance in Phase 3.

Phase 3: Comprehensive Evaluation Phase 3 focuses on analyzing the root causes of model failures while using the same reward formula as in Phase 2.

6.3.5 Ensuring Evaluation Validity

Dynamic challenge generation using the *Challenge Designer* (see Section 6.3.3) lowers the risk of benchmark memorization; however, it introduces two risks in return. Specifically, given a state's (c, d) pair:

- **Invalid challenge**: A challenge is invalid when it is off-concept or incorrectly defined relative to the selected state's concepts c. For example, challenges that do not test for the intended concept c (e.g., a sorting task labeled as dynamic programming), or infeasible or contradictory constraints (e.g., ambiguous input/output specifications, or evaluation criteria that cannot be met).
- **Miscalibrated difficulty**: A challenge is miscalibrated when its actual difficulty does not match the selected state's desired difficulty d. For example, a challenge is generated for an easy difficulty level that is actually hard or vice versa.

Below, we describe how *PrismBench*'s execution-based evaluation pipeline, search and aggregation mechanism with TD(0) and MCTS, alongside its multi-phase evaluation, mitigate these risks.

Execution Based Value Estimation

As detailed in Section 6.3.2, state selection is based on past observed performance. Once a state is selected, the *Challenge Designer* agent produces a challenge based on the state's (c, d), **independent** of other states. Afterwards, all rewards are derived from objective execution signals. Once the immediate *reward* for a state is calculated, the state's *value*

is determined using TD(0) (see Equation 6.2), and backpropagated through its ancestor states (see Equation 6.3). Therefore, an invalid or miscalibrated challenge will result in an immediate reward and subsequent state values that deviate from past observed performance. This deviation in state values informs state selection in the next iteration.

Search-time Self-correction

MCTS operates over a fixed MDP on (c, d) states, where transitions either add concepts or increase difficulty, and at each iteration, MCTS selects a state based on its current value estimate. As detailed above, when the observed reward is inconsistent with prior observations for that state or its ancestors (e.g., unexpectedly high success at a "hard" state or repeated failure at an "easy" state), the TD(0) update produces a noticeable deviation in the state's value. This, in turn, steers MCTS to revisit the invalid/miscalibrated state and its ancestors, which results in re-evaluation using fresh generations (given that challenges are generated at test time) and subsequent reward and value estimation calculations. Therefore, repeated revisits based on value estimations isolate noisy generations:

- If the original outlier was due to a one-off bad generation, subsequent evaluations will stabilize the value, and the branch is prioritized, resulting in MCTS exploring other branches.
- If the signal persists (meaning that previous observations were incorrect), then the value stabilizes at the new level and MCTS either expands adjacent states or prevents expansion when performance is systematically poor.

Therefore, states that produce inconsistent rewards are automatically detected for further exploration, and stable states' values converge. This feedback loop corrects for invalid/miscalibrated challenges at search time by continuous and iterative re-sampling and averaging over execution outcomes, and prevents unstable branches from affecting evaluation results.

Deterministic Challenge Selection

While the approaches described above lower the risks of evaluating models using invalid/miscalibrated challenges, as described in Section 6.2, there still exists the risk of the *Challenge Designer* itself being miscalibrated. Meaning that the agent used as the *Challenge Designer* has a different understanding of the difficulty levels in comparison to human preferences (i.e., the model's definition of a "hard" task corresponds to a "medium" task according to human preferences). In this case, the overall evaluation results become miscalibrated as the results of

a model's performance on "hard" tasks correspond to real-world performance on "medium" tasks.

While this does not affect the benchmarking methodology itself (given execution based estimation coupled with search time self-correction), it affects how the results are interpreted. As such, we mitigate this risk by fixing the *Challenge Designer* agent throughout the entire evaluation pipeline, regardless of the model being evaluated. In this manner, all models are evaluated on the same distribution of task difficulty. Furthermore, *PrismBench*'s modular design allows integration of task evaluation frameworks, such as TaskEval [313] or DyVal [314], to verify each challenge's validity. Finally, dynamic challenge generation can be entirely bypassed. Specifically, given a challenge bank with defined (c, d) properties for each challenge, such as CodeForces[1], the challenges for each node can be selected from said bank instead of being dynamically generated by an LLM.

6.3.6 Evaluation Metrics

We define four metric categories to capture distinct aspects of LLMs' code generation capabilities. These metrics provide a structured and thorough evaluation of a model's strengths, weaknesses, and solution strategies:

Structural Metrics

These metrics focus on the tree and how models perform in the search space. Node counts and depth distributions show where models struggle (persistent exploration) or succeed (rapid convergence), and tree growth patterns demonstrate how challenge complexity impacts performance. In this manner, we can provide fine-grained and detailed insights into model performance, behavior, and failures in order to address $\mathbf{RQc_3}$.

We denote the total number of nodes in the tree with $|N|$ and the depth of each node $n \in N$ with D_n. We track the distribution and connectivity of explored concepts through:

$$N(c) = \sum_{n \in \text{nodes}} 1_{[c \in \text{concepts}(n)]} \tag{6.15}$$

$$N(d) = \sum_{n \in \text{nodes}} 1_{[d \in \text{difficulties}(n)]} \tag{6.16}$$

with $N(c)$ and $N(d)$ being the number of times each concept and each difficulty was encountered throughout the entire tree. The node distribution across concepts and difficulties

[1] `https://codeforces.com`

provides a broad view of where the model succeeds and struggles: the greater the number of nodes associated with each concept and difficulty level, the less successful the model has been in addressing related challenges. Consequently, additional nodes were generated to better identify and isolate the problematic areas.

This is complemented by the branching factor at each node:

$$B(n) = \frac{\text{children}(n)}{|N|} \tag{6.17}$$

where children(n) is the number of children of node n and $|N|$ represents the total number of nodes. Nodes with higher branching factors have more children compared to the other nodes and, therefore, have been more challenging for the model.

The convergence rate $C(n)$ measures the stability of a model's performance at each node n by measuring the difference between consecutive TD values:

$$C(n) = |v_{t+k} - v_t| < \epsilon \text{ for } k = 1, \ldots, K \tag{6.18}$$

where v_t represents the node's TD value at attempt t. The convergence rate reflects how drastically the model's output changes between attempts. A phase is terminated when all nodes exhibit convergence rates below a predefined threshold ϵ for K consecutive attempts. A lower convergence rate indicates greater stability, meaning the model's performance has plateaued at node n. When this condition holds across all nodes in a phase, the phase is deemed to have been sufficiently explored, and the next phase begins.

Mastery Metrics

These metrics focus on the model's progress in understanding and applying concepts over the course of the benchmarking process. These metrics quantify performance stability as challenge complexity increases and success rates on challenges with combinations of concepts across benchmarking phases.

The primary measure of concept mastery is the success rate:

$$SR(c) = \frac{1}{N(c)} \sum_{n \in N} \text{success}(n) \tag{6.19}$$

where success(n) is 1 if the model has successfully passed the challenge at node n and 0 otherwise, as shown in Listing 15.

Similarly, we measure the model's success rate at each difficulty level:

$$SR(d) = \frac{1}{N(d)} \sum_{n \in N} \text{success}(n) \tag{6.20}$$

To understand the effort required for solving a challenge related to a concept c, we measure the average number of attempts regardless of success or failure:

$$A(c) = \frac{1}{N(c)} \sum_{n \in N(c)} \text{attempts}(n) \tag{6.21}$$

with attempts(n) representing the number of attempts made at node n as shown in Listing 15.

These three metrics alongside each other, indicate how well the model performs in solving challenges for each specific concept/difficulty with the average number of attempts indicating how many times the model encountered errors while solving the challenge. High success rates and low number of attempts indicate a high capability (the challenge was solved with a low number of errors and attempts) while lower success rates and higher number of attempts indicate struggles in solving challenges with that specific concept/difficulty.

Performance Metrics

Performance metrics build upon the mastery metrics and assess the model's performance across different concepts and difficulty levels by providing a granular understanding of the model's capabilities using challenge success rates, the number of interventions required to fix the model's code, and problem-solving efficiency across concepts and difficulty levels.

The fixer intervention rate indicates when the model requires external help and is unable to solve the challenge:

$$F(c) = \frac{1}{N(c)} \sum_{n \in N(c)} 1_{[I_{fixer}(n)]} \tag{6.22}$$

with I_{fixer} being 1 if the *Problem Fixer* agent was used at each node n and 0 otherwise.

As shown in Algorithm 13 the *Problem Fixer* agent is only used when the model fails in all of its attempts to solve the challenge. This is caused by either incorrect solutions or incorrect tests. Therefore, we can measure the model's program repair capabilities for each concept by tracking whether the use of *Problem Fixer* resulted in success:

$$R(c) = \frac{\sum_{n \in N(c)} 1_{[\text{success}(n)]}}{\sum_{n \in N(c)} 1_{[I_{\text{fixer}}(n)]}} \tag{6.23}$$

where success(n) is 1 if the model has successfully passed the challenge at node n and 0 otherwise after the *Problem Fixer* intervention at node n.

Diagnostic Metrics

These metrics reveal behavioral patterns through solution analysis (preferred coding patterns), error categorization (common failure modes), and test set evaluation (correct tests, testing for corner cases, etc). They capture how the model succeeds or fails in specific scenarios and allow for identifying and characterizing the model's behavior (how it solves the challenges and how it fails).

The distribution of solution patterns across concepts shows how many times the model has used a specific solution for each concept c:

$$SP(p, c) = \frac{\text{count}(p, c)}{N(P)} \tag{6.24}$$

with $N(P)$ being the number of identified patterns throughout the entire tree and patterns indicating algorithmic approaches, data structure usage, and implementation patterns.

Pattern effectiveness quantifies which solutions the model executes successfully, helping identify its preferred problem-solving strategies:

$$PE(p) = \frac{\sum_{n \in N(p)} 1_{[\text{success}(n)]}}{N(P)} \tag{6.25}$$

Conversely, using failure rate $(1 - success(n))$ instead of success rate quantifies the patterns the model struggles with the most.

Test validation scores test suite quality:

$$TV(v, c) = \frac{\text{count}(v, c)}{N(V)} \tag{6.26}$$

where $N(V)$ is the number of identified validation issues throughout the entire tree with v being an identified validation issue with validation issues including analyses on missing, incorrect, coverage, and corner case issues for each generated test suite for each concept c.

Error pattern distribution by concept shows where exactly the model has failed in solving the challenges related to that concept c:

$$EP(e, c) = \frac{\text{count}(e, c)}{N(E)} \tag{6.27}$$

where $N(E)$ is the number of identified errors throughout the entire tree with e being an error that was raised during the execution of the program.

6.4 Experimental Design

To show *PrismBench*'s effectiveness, we evaluate 8 LLMs on their code generation, test suite creation, and program repair capabilities: **GPT4o** (4o), **GPT4o-mini** (4o-M) [315], **GPT-OSS-20b** (GPT-OSS) [316], **Llama3.1-8b** (L-8b), **Llama3.1-70b** (L-70b), **Llama3.1-405b** (L-405b) [317], **Llama4-Scout** (L4S) [318], and **DeepSeekV3** (DS3) [148]. For our experiments, we use LeetCode (LC) [319] style programming challenges and 4o-M as the *Challenge Designer* to create problems based on each node's concepts and difficulty levels for **all** models under evaluation. The test generation, code generation, and repair tasks are performed by the models under evaluation, while 4o is used for analyzer agents in Phase 3. All reported results are averaged over 3 independent benchmarking runs for all models under study.

For all LLMs under study, the concepts are chosen similarly to the fundamental concepts of computer science in LC, with difficulty levels of "very easy", "easy", "medium", "hard", and "very hard", the same as the difficulties of LC challenges, which we describe in detail below.

6.4.1 Concepts

Here, we provide a concise explanation of each concept and what we expect the models to achieve in tasks involving these concepts.

- **Loops**: A loop is a control structure that repeatedly executes a block of code as long as a specified condition is true. Examples include `for`, `while`, and `do-while` loops. As such the models should:
  - Correctly implement loops to traverse data structures or repeat operations.
  - Optimize loop usage for efficiency and avoid common pitfalls such as infinite loops.
- **Conditionals**: Conditionals are control structures that execute specific code blocks based on boolean conditions. Examples include `if`, `else`, and `else if` statements. We expect the model to:
  - Accurately implement conditionals to manage decision-making logic.
  - Handle edge cases and ensure logical correctness when combining multiple conditions.

- **Functions**: Functions are reusable blocks of code that perform a specific task, defined by a name, parameters, and a return value. The models should:
  - Design modular and reusable functions.
  - Handle parameter passing and scope effectively.
- **Data Structures**: Data structures organize and store data to facilitate efficient access and modification. Examples include arrays, linked lists, stacks, queues, and trees. The models should:
  - Choose appropriate data structures for given problems.
  - Implement and manipulate data structures accurately and handle edge cases.
- **Algorithms (logic)**: Step-by-step procedures for solving problems or performing computations. As such, the models should:
  - Devise efficient algorithms to address specified problems.
  - Optimize time and space complexity, demonstrating an understanding of computational trade-offs.
- **Error Handling**: Error handling involves detecting, managing, and responding to runtime errors. As such, the models should:
  - Implement robust error-handling mechanisms, including exception handling and validation.
- **Recursion**: Recursion is a technique where a function calls itself to solve a problem by breaking it into smaller sub-problems. As such, the models should:
  - Correctly implement recursive functions, ensuring termination through base cases.
  - Optimize recursion to avoid excessive memory usage and stack overflow issues.
- **Sorting**: Sorting involves arranging data in a specific order, such as ascending or descending, such as quicksort, mergesort, and bubble sort. As such, the models should:
  - Implement sorting algorithms correctly and select appropriate algorithms for the given data size and constraints.
- **Searching**: Searching involves finding specific elements in a dataset, such as linear search, binary search, and hash-based lookups. As such, the models should:

  - Apply efficient search techniques suited to the dataset's structure.
  - Ensure correctness and handle cases where the element is not present.

- **Dynamic Programming**: Dynamic programming is a technique for solving complex problems by breaking them into overlapping sub-problems and solving each sub-problem only once. We expect the models to:
  - Develop dynamic programming solutions to problems requiring optimization.
  - Demonstrate the ability to use memoization or tabulation correctly.

These concepts are foundational to CS and cover the essential problem-solving skills required to implement solutions and tests for a problem. By benchmarking models on these concepts, we aim to assess their ability to generalize to unseen tasks based on single concepts and concept combinations critical for coding and reasoning. The concepts for benchmarking are modifiable, meaning that they can be changed to any desired topic, allowing *PrismBench* to be used in more specific scenarios and subjects (e.g., instead of foundational concepts, implementation patterns and challenges closer to LC challenges such as "Two Sum", "Valid Sudoku", etc. can be chosen).

6.4.2 Combination of Concepts

As we detailed in Section 6.3.2, in real-world programming scenarios, the implementation of solutions rarely requires implementing isolated, single concepts. Instead, they require the integration of multiple concepts to address complex problems effectively. For example, developing a functional application often involves combining loops for iteration, conditionals for decision making, and data structures to organize information. In addition, advanced challenges frequently require recursion, algorithms for processing logic, and error handling to ensure that the program does not fail when it encounters unexpected inputs or conditions.

Therefore, to simulate real-world programming scenarios, *PrismBench* generates challenges that combine these core concepts into unified problems. This allows us to evaluate a model's capabilities to synthesize knowledge across programming concepts. For example, a single problem might require using dynamic programming alongside data structures for optimal solutions or using sorting and searching techniques to manage/query datasets. This approach ensures that the model can demonstrate competency in scenarios requiring cross-concept integration. As such, failure to solve problems involving multiple concepts is an indicator of deficiencies in one or more of the constituent concepts. Such failures signal areas where the

model struggles to integrate distinct methodologies or lacks a deep understanding of specific concepts. For instance, if a model fails a task combining functions and error handling, it might reflect difficulties in managing exceptions within modularized code. In this manner, *PrismBench* can investigate these failures further by identifying the exact concepts or combinations responsible for failures.

Alongside combining concepts, we also use a range of difficulty levels: very easy, easy, medium, hard, and very hard in order to perform fine-grained analysis of the model's capabilities. This enables us to assess performance not only on single concepts and their combinations but also on different complexities of these problems. For example, a model might perform well on easier problems related to a concept or group of concepts but fail on medium or hard ones, revealing limitations in its ability to scale solutions to more challenging scenarios.

By probing models across a variety of concept combinations and difficulty levels, we gain a comprehensive understanding of their strengths and weaknesses and gain valuable insights into their overall code generation capabilities by pinpointing root causes and systematically evaluating a model's limitations.

6.4.3 Experiment Settings and Reproducibility

In this section, we report the configuration values for all global and phase-specific parameters used throughout *PrismBench*'s multi-phase evaluation pipeline for the experiments reported in our study. These values were kept fixed across experiments on all models to ensure comparability and reproducibility of the results.

Benchmarking parameters Table 6.1 lists all the parameters used in our study. We organize *PrismBench*'s tunable parameters into a set of global settings that control the overall search and value-estimation behavior, alongside phase-specific parameters that control the reward functions and exploration policies at each stage of the evaluation. All values listed below were held constant across the experiments reported in our work.

Agent configurations Table 6.2 shows the agent configurations used during each model's evaluation. As outlined in Section 6.4, we use 4o-M as the *Challenge Designer* across all experiments. For each model being benchmarked, we use that model as the *Test Generator*, *Problem Solver*, and *Problem Fixer* agents. To avoid bias in error analysis, we use 4o as the *Test Validator*, *Test Error Analyzer*, and *Solution Pattern Analyzer* for all models except when benchmarking 4o itself. In that case, we use L-405b as the analyzer agent to prevent bias.

Table 6.1 Tunable parameters

Description	Parameter	Value
General		
Discount factor (Eq. 6.3)	γ	0.9
Learning rate (Eq. 6.2)	α	0.9
Exploration constant (Eq. 6.7)	C	1.414
Number of convergence checks	–	5
Phase 1		
Node value threshold (Eq. 6.5)	θ_p	0.4
Value delta threshold (Sec. 6.3.4)	$\Delta(v)$	0.3
Random exploration probability (Eq. 6.11)	ϵ_1	0.2
Penalty per failure (Eq. 6.14)	P_{failure}	2
Penalty per error (Eq. 6.14)	P_{error}	3
Penalty per attempt (Eq. 6.14)	P_{attempt}	1
Penalty for using Problem Fixer (Eq. 6.14)	P_{fixer}	5
Max depth (Eq. 6.5)	$d_{\max}$	5
Phase 2		
Node value threshold (Eq. 6.5)	θ_p	0.6
Value delta threshold (Sec. 6.3.4)	$\Delta(v)$	0.1
Random exploration probability (Eq. 6.12)	ϵ_2	0.1
Max depth (Eq. 6.5)	$d_{\max}$	10
Phase 3		
Number of variations for each node	–	5
Node value threshold (Eq. 6.5)	θ_p	0.5

6.4.4 Benchmarking cost

Evaluating each node in the search tree requires independent calls to each agent, with the process being dependent on the model's performance. For every node, we use one query for the *Challenge Designer* to generate the challenge, one for the *Test Generator*, and one for the *Problem Solver*. If the model succeeds on the first attempt, the evaluation ends with these three queries. However, if the solution fails, up to three additional queries are used for the *Problem Solver* for iterative repair attempts. If still unsolved, one additional query is used for *Problem Fixer* to repair the solution. As detailed in Section 6.3.4, nodes may also be revisited up to five times if their children consistently receive low rewards (convergence checks up to 5 times). Finally, for each node generated in Phase 3, three more queries are used for the *Test Validator*, *Test Error Analyzer*, and *Solution Pattern Analyzer* agents.

Table 6.2 Experiments configurations

Agent	**4o**	**4o-M**	**L-8b**	**L-70b**	**L-405b**	**L4S**	**DS3**	**GPT-OSS**
Challenge Designer	4o-M	4o-M	4o-M	4o-M	4o-M	4o-M	4o-M	4o-M
Test Generator	4o	4o-M	L-8b	L-70b	L-405b	L4S	DS3	GPT-OSS
Problem Solver	4o	4o-M	L-8b	L-70b	L-405b	L4S	DS3	GPT-OSS
Problem Fixer	4o	4o-M	L-8b	L-70b	L-405b	L4S	DS3	GPT-OSS
Test Validator	L-405b	4o	4o	4o	4o	4o	4o	4o
Test Error Analyzer	L-405b	4o	4o	4o	4o	4o	4o	4o
Solution Pattern Analyzer	L-405b	4o	4o	4o	4o	4o	4o	4o

Therefore, the per-node benchmarking cost in our framework ranges from a minimum of 3 queries (one-shot success, no retries) to a maximum of 38 queries (persistent failures, repeated convergence checks, and full diagnostic analysis). On average, throughout all the trees in our experiments, a single node has triggered 6 queries.

The total query count for a full benchmarking run is dependent on the model's capabilities and provider costs(if using APIs). For the experiments conducted in this study, the number of queries and cost per model, averaged over 3 independent runs, are as follows:

- For 4o, a full run triggered 1,153 queries with each run costing approximately US$20.
- For 4o-M, a full run triggered 961 queries, with each run costing approximately US$10.
- For GPT-OSS, a full run triggered 1363 queries, with each run costing approximately US$16.
- For L-405b, a full run triggered 1,094 queries, with each run costing approximately US$24.
- For L-70b, a full run triggered 454 queries, with each run costing approximately US$14.
- For L-8b, a full run triggered 183 queries, with each run costing approximately US$8.
- For L4S, a full run triggered 480 queries, with each run costing approximately US$10.
- For DS3, a full run triggered 884 queries, with each run costing approximately US$13.

Reducing Benchmarking Costs

As mentioned above, evaluating each node in the search tree requires 35 calls in the worst-case scenario. While *PrismBench* significantly lowers the sampling requirements for comprehensive evaluation of LLM capabilities, the workflow of using all 7 agents for each node

can become computationally and financially expensive. As such, *PrismBench* allows for a lightweight version of the benchmarking process by using challenges generated from previous runs and skipping diagnostic metric calculation. Specifically:

- By using a bank of generated challenges from previous runs, the *Challenge Designer* agent can be turned off.
- The 3 analyzer agents (*Test Validator*, *Test Error Analyzer*, and *Solution Pattern Analyzer*) are only used for analysis of the benchmarking results in order to compute the diagnostic metrics described in Section 6.3.6 and other metrics are not dependent on LLM analysis. Therefore, they can be turned off in case diagnostic metrics are not required.

The combination of these two solutions reduces the number of LLM calls per node in the worst-case scenario from 38 to 15, and reduces computational and financial costs of the benchmarking process by up to 50%.

6.5 Results and Analysis

In this section, we present the experimental results and analysis of our dynamic benchmarking approach, as described in Section 6.3. We begin with a comparative analysis of the eight LLMs introduced in Section 6.4 by evaluating their code generation performance using the metrics defined in Section 6.3.6. Afterwards, we provide a fine-grained breakdown of how each model performed across different dimensions in Section 6.5.2. Finally, in Section 6.5.3, we analyze the effects of model scale (i.e., number of parameters) and how it impacts code generation capabilities.

6.5.1 Comparative Analysis

In this section, we provide a comparative analysis of evaluation results across the four metric categories discussed in Section 6.3.6 for the models under study.

Structural Metrics

Figure 6.4 compares the tree growth per depth in Phase 1. At the very beginning of the search, the tree is populated with initial nodes in order to provide the same starting point for all models and allow comparison between their performance as the evaluation process

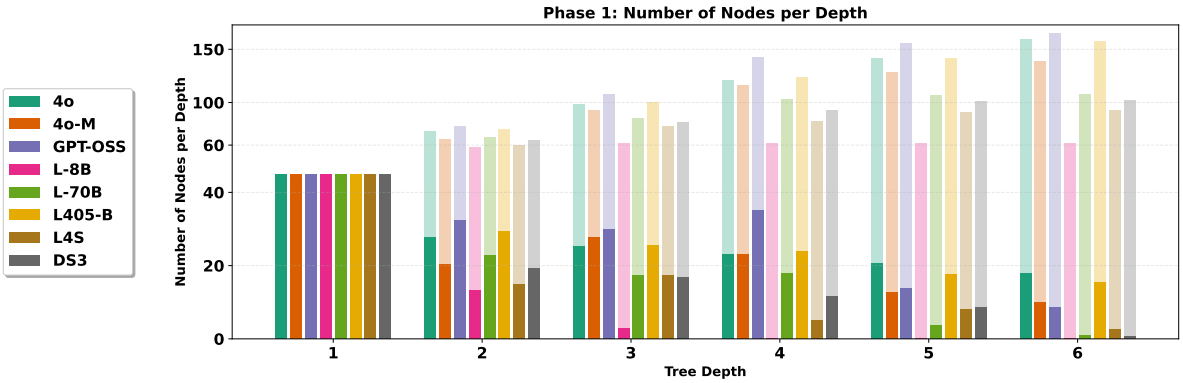


Figure 6.4 **Tree growth across models throughout Phase 1** The bars represent node counts by tree depth, and the shaded bars represent the cumulative number of nodes per depth across the tree in *Phase 1*.

continues (see Section 6.3.4). Phase 1's reward function prioritizes task success and difficulty-weighted exploration (see Section 6.3.4). Therefore, the shaded bars for each model quantify how effectively they sustain problem-solving capability as challenges become more complex (i.e., we go deeper in the tree): higher number of nodes indicates broader exploration and lower failures. For instance, 4o and GPT-OSS achieve more than 150 nodes in Phase 1, demonstrating robust handling of complex challenges (e.g., multi-concept and high-difficulty tasks), while L-8b stalls at 60 nodes, failing beyond basic concepts and easy difficulties (depth<4).

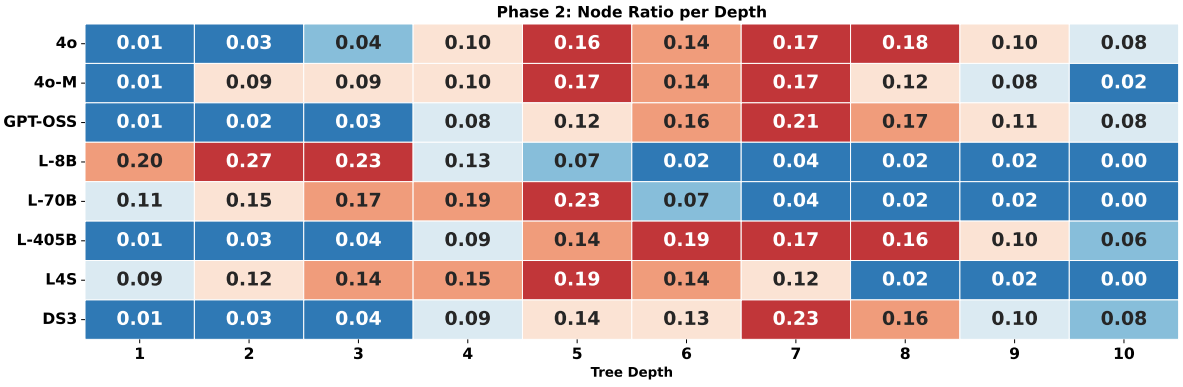


Figure 6.5 **Node ratio by depth across models throughout Phase 2.** Each cell shows the ratio of nodes in the tree at each depth, indicating relative search focus across the tree in *Phase 2*.

Phase 2 uses low-scoring Phase 1 nodes to generate targeted challenges, prioritizing *task failure* and *repeated attempts*. Therefore, the ratio of generated nodes per depth, as shown

in Figure 6.5, reveals where models struggle: higher ratios at shallower depths imply difficulty with simpler challenges, while increasing ratios at greater depths demonstrate stronger problem-solving capability at complex challenges. We can observe that even though 4o-M has a similar number of nodes than L-405b at the end of Phase 1 (142 vs 151 nodes), it struggles with complex challenges in Phase 2, while L-405b demonstrates a more consistent exploration of the tree and has a higher ratio of nodes compared to 4o-M at the end of Phase 2. Furthermore, we can observe that GPT-OSS has a similar problem-solving capability to 4o and L-405b despite being much smaller in scale, which we discuss in detail in Section 6.5.2.

Performance Metrics

Table 6.3 summarizes model capability analysis at the end of the benchmark, with values showing *failure rates* across concepts and difficulty levels. *PrismBench* dynamically explores the search space to find challenging areas for the model, then focuses on these areas to uncover root causes of failure. The primary operational capability for each concept is determined by the ratio of nodes (concept-difficulty pairs) explored in the search tree and their average failure rates over 3 independent runs.

Table 6.3 **Model capability analysis by concept and difficulty**. Values represent failure rates (higher = more challenging). Colors indicate performance: green (good) to red (poor). † indicates primary operational difficulty level (most number of nodes), ✓indicates mastered concepts (failure rate < 0.01), and ✗ indicates concepts beyond current capability (failure rate > 0.99). 95% CI broken down by concept and difficulty reported in Tables 6.4 and 6.5

	Very Easy/Easy								Medium								Hard/Very Hard							
Concept	4o	4o-M	GPT-OSS	L405	L70	L8	L4S	DS3	4o	4o-M	GPT-OSS	L405	L70	L8	L4S	DS3	4o	4o-M	GPT-OSS	L405	L70	L8	L4S	DS3
Algorithms	✓	✓	✓	0.50	0.81†	0.78†	0.30	✓	✓	0.53	0.39	✓	0.91†	0.92†	0.62†	✓	0.66†	0.89†	0.40	0.73†	✗	✗	0.70†	0.67†
Conditionals	✓	✓	✓	✓	0.81†	0.79†	✓	✓	0.75†	✓	0.33	0.85†	0.92†	✗	0.41	0.32	0.67†	0.88†	0.44	0.64†	✗	✗	0.48	0.71†
Data Struct.	✓	0.60	✓	✓	0.80†	0.77†	0.29	✓	0.75†	0.53	0.29	0.85†	0.98†	✗	0.50	0.48	0.67†	0.88†	0.40	0.68†	✗	✗	0.35	0.77†
Dyn. Prog.	✓	0.60	0.29	0.50	0.77†	0.78†	0.62†	✓	✓	0.53	0.32	0.85†	0.88†	✗	0.78†	0.53	0.67†	0.89†	0.58	0.71†	✗	✗	0.54	0.89†
Error Hand.	✓	✓	0.26	✓	0.82†	0.78†	✓	✓	0.75†	0.53	✓	0.85†	✗	✗	0.39	✓	0.67†	0.89†	0.38	0.71†	✗	✗	0.87†	0.30
Functions	✓	0.60	✓	✓	0.82†	0.79†	✓	✓	0.75†	0.53	✓	0.85†	✗	0.92†	0.69†	✓	0.66†	0.90†	0.44	0.70†	✗	✗	0.57	0.51
Loops	✓	✓	✓	✓	0.81†	0.79†	0.33	✓	✓	0.53	✓	0.85†	0.81†	0.92†	0.70†	✓	0.67†	0.88†	0.29	0.71†	✗	✗	0.38	0.86†
Recursion	✓	✓	✓	0.50	0.79†	0.77†	✓	✓	✓	0.53	✓	0.85†	✗	✗	0.60	✓	0.66†	0.89†	0.35	0.68†	✗	✗	0.57	0.72†
Searching	✓	✓	✓	0.50	0.80†	0.78†	0.32	✓	0.75†	0.53	✓	0.85†	✗	✗	0.52	✓	0.67†	0.89†	0.28	0.70†	✗	✗	0.67†	0.54
Sorting	✓	0.60	✓	✓	0.79†	0.78†	0.26	✓	✓	0.53	✓	0.85†	0.92†	0.98†	0.39	0.42	0.66†	0.89†	0.30	0.71†	✗	✗	0.47	0.70†

4o shows no failures on easy tasks, demonstrating strong basic programming skills. However, performance drops at higher difficulty levels, especially for "dynamic programming" and "data structures", indicating limitations in handling programming challenges that require in-depth reasoning. L-405b fails on some easy challenges, but generally has lower failure rates on easy and medium tasks. Similar to 4o, it struggles with hard/very hard challenges that require integration of multiple concepts, such as "dynamic programming", "algorithms", and "functions". 4o-M has higher failure rates overall among the top-performing models, especially for

challenges requiring compositional reasoning, such as “loops”, “functions”, “conditionals”, and “recursion”. These failures are more common when concepts are combined (e.g., loops with conditionals), as shown in Figure 6.20 and discussed in detail in Section 6.5.2.

Both GPT-OSS and DS3 display higher levels of capability than 4o and L-405b, achieving lower failure rates on medium and hard/very hard tasks. While this is indicative of better coding capability across concepts and difficulty levels, the majority of GPT-OSS’s and DS3’s successes, are a result of relying on Python’s built-in functions and standard libraries (e.g., using `sort` on arrays instead of implementing the sorting function, using `itertools` to iterate through multiple arrays at once instead of implementing the functionality, etc.) **despite** task instructions that prohibit such solutions. Therefore, while GPT-OSS and DS3 are more capable in solving challenges in contrast to 4o and L-405b, they are less capable in instruction following and adhering to specified requirements, which are revealed through the diagnostic metrics (error patterns and test validations) which we discuss in detail in Section 6.5.2.

L4S’s higher success rates are a result of it reaching much fewer nodes at higher depths compared to both 4o and L-405b as shown in Figure 6.5, which indicates L4S’s relative search focus, and Figure 6.7, which displays the success rates weighted by the number of node visits. On the other hand, L-70b and L-8b show different patterns: L-70b struggles even with easy challenges and fails more as difficulty increases, indicating a limited capacity for complex challenges. L-8b has high failure rates across all concepts and difficulties, indicating limitations in basic code generation capability. We provide a more detailed analysis of each model’s performance and the effects of model scale in Sections 6.5.2 and 6.5.3.

Mastery Metrics

Figures 6.6 and 6.7 present *success rates* by concept and difficulty for the top models across Phases 1 and 2 *weighted* by the number of node visits for each concept at each difficulty level (higher number indicates more success at fewer attempts). While Performance metrics capture overall challenge outcomes (i.e., success/failure), mastery metrics highlight which concepts each model handles well and where they struggle. By mapping success rates to each concept-difficulty pair and weighing them by the number of visits to associated nodes, we can pinpoint common failure modes and determine each model’s limits.

As shown in Figure 6.6, GPT-OSS outperforms all models on medium and hard/very hard difficulty challenges, particularly in “loops”, “searching”, “sorting”, “data structures”, and “algorithms” with a marginally lower capability in function-heavy composition and “dynamic programming”. 4o struggles with compositional reasoning: challenges that require combining “algorithms”, “data structures”, or “dynamic programming”, especially when multiple

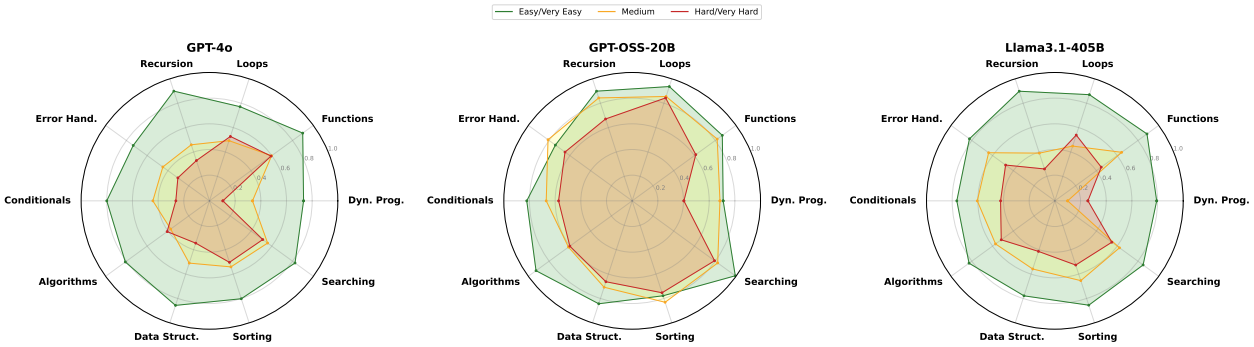


Figure 6.6 **Concept success rate analysis per difficulty for 4o, GPT-OSS, and L-405b**. Green: very easy/easy, Yellow: medium, Red: hard/very hard. The radial axis represents the success rate (between 0 and 1). Each axis corresponds to a programming concept. Higher values indicate better performance.

function calls, nested conditionals, or multiple levels of recursion are required. L-405b shows strong performance on very easy/easy challenges and similar performance to 4o on medium difficulty control flow and data structure tasks. However, similar to 4o, it struggles with complex challenges such as "dynamic programming" or "recursion" on hard/very hard difficulty levels.

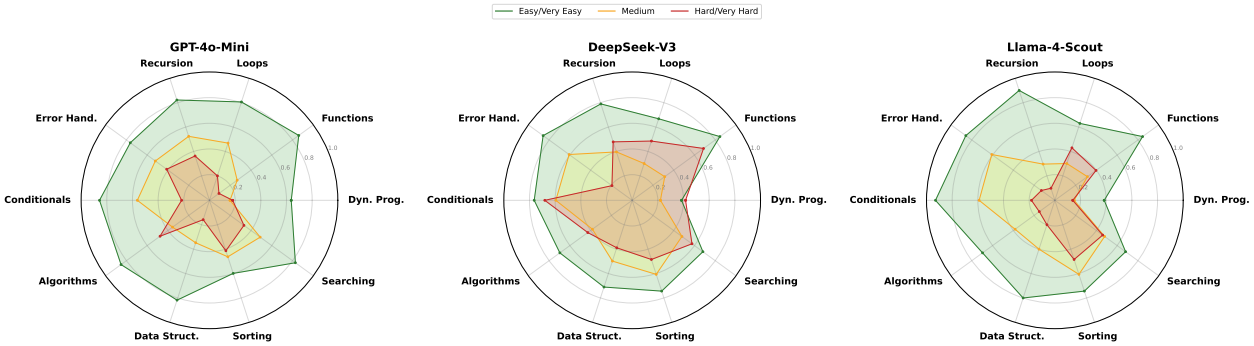


Figure 6.7 **Concept success rate analysis per difficulty for 4o-M, DS3, and L4S**. Green: very easy/easy, Yellow: medium, Red: hard/very hard. The radial axis represents the success rate (between 0 and 1). Each axis corresponds to a programming concept. Higher values indicate better performance.

4o-M shows high capability on easy/very easy challenges, but its success rates degrade as difficulty levels increase, specifically across challenges that require "recursion", nested conditionals, and "dynamic programming". On the other hand, while DS3 shows high success rates on easy and medium level difficulties, its performance sharply degrades on challenges involving "dynamic programming", "error handling", and loop-intensive compositions. As

mentioned in Section 6.5.1 and shown in Figure 6.7, even though L4S achieves high success rates on hard/very hard difficulty challenges, it is incapable of solving them consistently because it visits far fewer nodes with such levels of difficulty. As such, when weighted by the number of node visits, L4S's success rates are much lower compared to other top models. We include a more detailed analysis of concept combinations and their effects on model performance in Section 6.5.2.

Diagnostic Metrics

Figures 6.8 and 6.9 show the success ratios of the top-performing models for the four highest failure rate concepts from Table 6.3, grouped by the top three programming patterns found in the solutions of each model.

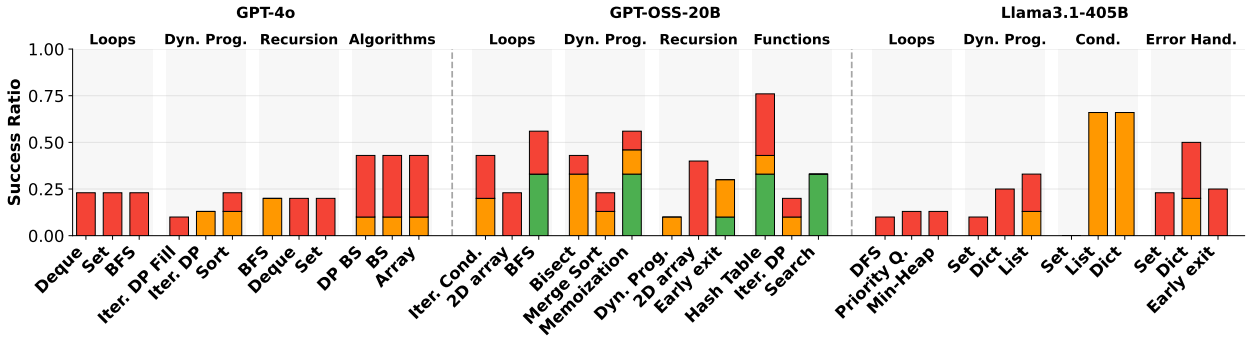


Figure 6.8 **Success ratios for the most challenging programming patterns for 4o, GPT-OSS, and L-405b**, grouped by the 3 most challenging concepts for each model. Stacked bars show performance by difficulty. Green: medium, orange: hard, red: very hard. Taller bars indicate better performance.

As shown in figure 6.8, 4o struggles significantly with "dynamic programming", even when the concept is not explicitly in the challenge. In contrast to the other models under study, the majority of GPT-OSS's solutions involve multiple nested function definitions (i.e., defining functions inside other functions), which, when combined with GPT-OSS's over-reliance on Python's standard libraries to either create hash tables or search through the inputs in intermediate steps, result in failures. L-405b shows the lowest success ratios for simple "data structures" and "tree/graph traversal". In contrast to the other top models, our analysis shows that L-405b's failures are not due to a lack of understanding of the problem itself but from failures in instruction-following and programming syntax. L-405b's failed solutions are often implemented using built-in data types (set, list, dict, etc.) and while the logic and pseudocode are often correct, L-405b frequently makes errors such as hallucinating keys in built-in types (using incorrect attributes), misplacing code snippets (calling a variable

before defining it), or failing to follow the system prompt's format, which lead to immediate rejection of solutions by the framework.

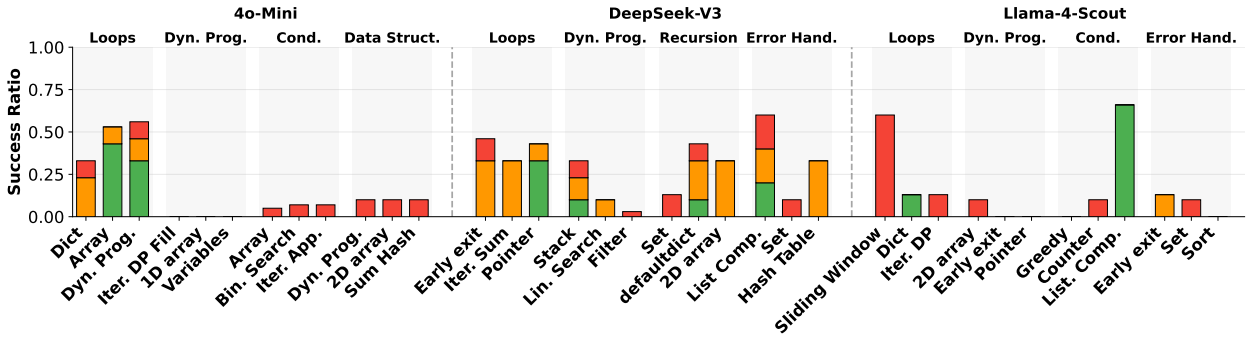


Figure 6.9 **Success ratios for the most challenging programming patterns for 4o-M, DS3, and L4S**, grouped by the 3 most challenging concepts for each model. Stacked bars show performance by difficulty. Green: medium, orange: hard, red: very hard. Taller bars indicate better performance.

On the other hand, 4o-M consistently fails in challenges involving composite problems (combinations of multiple concepts), "complex data structures", or "dynamic programming", regardless of how it attempts to solve the challenge. Similar to GPT-OSS, DS3 relies on Python's standard libraries and built-in functions in order to solve the challenges; however, the majority of DS3's failures are a result of incorrect input type and shape estimation (e.g., assuming the input will be a 1D array and failing when the input is higher-dimensional). Compared to 4o and L-405b, DS3 extrapolates requirements the most without considering other possibilities, which is the main root cause behind its consistent failures for solving challenges involving "error handling". Finally, L4S struggles the most with challenges requiring iterative processing of inputs or intermediate results. We provide a comprehensive breakdown of these observations through per-model analysis and cross-model comparison in Section 6.5.2.

6.5.2 Detailed Analysis of Results

Tables 6.4 and 6.5 show the average success rate and average intervention rate for each of the models under study, across concepts and difficulties, respectively. The metrics presented here are averaged from the values throughout the entire tree at the end of the benchmarking process, for 3 independent benchmarking runs for each model, and are not phase-specific.

Looking at the performance data across all models, we observe a clear hierarchy in both success rates and the number of interventions. Starting with the model performance by

Table 6.4 **Model performance by difficulty**. Colors indicate performance: green (good) to red (poor). Higher values for intervention rates indicate more usage of the *Fixer* agent. Each cell shows the mean value over 3 runs ± margin of error, calculated at a 95% confidence interval.

Difficulty	4o		DS3		GPT-OSS		L405b	
	Avg Succ. Rate	Avg Inter.	Avg Succ. Rate	Avg Inter.	Avg Succ. Rate	Avg Inter.	Avg Succ. Rate	Avg Inter.
Very easy	0.83 ± 0.11	4.67 ± 0.02	0.83 ± 0.01	3.33 ± 0.12	0.89 ± 0.05	1.33 ± 0.04	0.83 ± 0.04	3.67 ± 0.03
Easy	0.73 ± 0.12	2.00 ± 0.02	0.64 ± 0.11	1.00 ± 0.12	0.85 ± 0.09	3.33 ± 0.02	0.72 ± 0.10	2.33 ± 0.07
Medium	0.42 ± 0.01	1.00 ± 0.01	0.51 ± 0.02	0.33 ± 0.04	0.75 ± 0.04	4.33 ± 0.09	0.39 ± 0.10	1.33 ± 0.01
Hard	0.33 ± 0.09	2.00 ± 0.04	0.64 ± 0.12	0.33 ± 0.11	0.60 ± 0.12	2.67 ± 0.09	0.46 ± 0.07	1.00 ± 0.04
Very hard	0.29 ± 0.08	2.00 ± 0.10	0.28 ± 0.05	0.00 ± 0.01	0.49 ± 0.03	9.67 ± 0.12	0.29 ± 0.07	2.50 ± 0.06
	4o-M		L4S		L70b		L8b	
	Avg Succ. Rate	Avg Inter.	Avg Succ. Rate	Avg Inter.	Avg Succ. Rate	Avg Inter.	Avg Succ. Rate	Avg Inter.
Very easy	0.83 ± 0.10	1.00 ± 0.03	0.84 ± 0.02	1.12 ± 0.04	0.42 ± 0.08	3.00 ± 0.05	0.19 ± 0.11	1.67 ± 0.06
Easy	0.63 ± 0.03	1.00 ± 0.09	0.66 ± 0.03	1.00 ± 0.03	0.22 ± 0.01	0.00 ± 0.05	0.22 ± 0.11	1.00 ± 0.03
Medium	0.35 ± 0.02	1.00 ± 0.01	0.51 ± 0.08	1.33 ± 0.08	0.16 ± 0.09	0.00 ± 0.01	0.03 ± 0.06	0.00 ± 0.10
Hard	0.21 ± 0.11	1.50 ± 0.03	0.33 ± 0.12	2.33 ± 0.04	0.21 ± 0.12	1.00 ± 0.02	0.11 ± 0.06	0.00 ± 0.09
Very hard	0.24 ± 0.08	1.00 ± 0.10	0.29 ± 0.01	0.00 ± 0.10	0.20 ± 0.07	1.00 ± 0.05	0.07 ± 0.06	1.00 ± 0.06

difficulty level, there's a consistent degradation in success rates as difficulty increases across all models. As expected, the "very easy" difficulty level shows the highest success rates for all models. The success rates steadily decline to much lower values at "very hard" difficulties. The success rates of L-70b and L-8b even on the "very easy" difficulty level compared to the other models, already indicate the limited capability of these models given the number of their parameters, which we discuss in depth in 6.5.3.

In terms of concept mastery, we see varying performance across models. 4o performs best on "functions" and "searching" challenges, while struggling with "dynamic programming". 4o-M shows more consistent performance across concepts but with lower overall success rates. L-405b demonstrates solid capabilities on "error handling" and "searching" challenges while also struggling with "conditionals", and similar to 4o and 4o-M, on "dynamic programming". The smaller Llama models (L-70b and L-8b) show significantly lower success rates across all concepts, with L-8b particularly struggling with success rates mostly below 0.20.

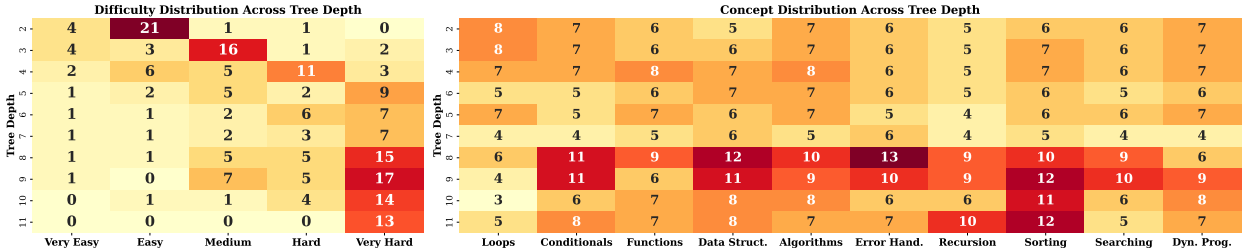
Both 4o and L-405b show notably high intervention rates, especially at the "very easy" difficulty level (4.67 and 3.67, respectively). This is particularly interesting given that these models also maintain high success rates. Investigating node distributions helps explain these patterns with Figures 6.10a and 6.10b displaying the distribution of nodes in each depth per concept and difficulty for 4o and L-405b, respectively. Both models quickly progress beyond "very easy" difficulty challenges, as evidenced by their node distributions (15 and 14 nodes

Table 6.5 **Model performance by concept**. Colors indicate performance: green (good) to red (poor). Higher values for intervention rates indicate more usage of the *Fixer* agent. Each cell shows the mean value over 3 runs ± margin of error, calculated at a 95% confidence interval.

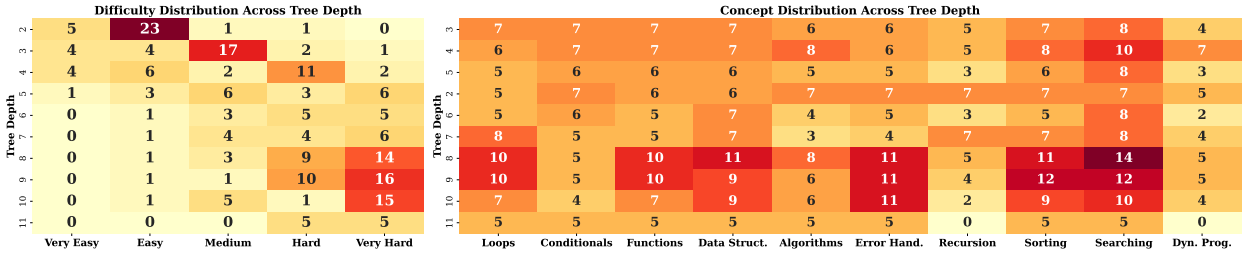
Concept	4o		DS3		GPT-OSS		L405b	
	Avg Succ. Rate	Avg Inter.	Avg Succ. Rate	Avg Inter.	Avg Succ. Rate	Avg Inter.	Avg Succ. Rate	Avg Inter.
Loops	0.51 ± 0.10	2.50 ± 0.03	0.47 ± 0.02	0.33 ± 0.04	0.73 ± 0.08	5.00 ± 0.05	0.48 ± 0.11	3.00 ± 0.06
Conditionals	0.42 ± 0.03	4.33 ± 0.09	0.69 ± 0.03	0.00 ± 0.03	0.65 ± 0.01	4.33 ± 0.05	0.43 ± 0.11	2.33 ± 0.03
Data Struct.	0.43 ± 0.02	4.33 ± 0.01	0.52 ± 0.08	0.00 ± 0.08	0.68 ± 0.09	5.00 ± 0.01	0.43 ± 0.06	3.33 ± 0.10
Algorithms	0.49 ± 0.11	4.50 ± 0.03	0.55 ± 0.12	1.00 ± 0.04	0.66 ± 0.12	4.67 ± 0.02	0.50 ± 0.06	3.00 ± 0.09
Dyn. Prog.	0.29 ± 0.08	2.50 ± 0.10	0.32 ± 0.01	0.00 ± 0.10	0.56 ± 0.07	4.33 ± 0.05	0.42 ± 0.06	3.33 ± 0.06
Error Hand.	0.49 ± 0.08	1.67 ± 0.01	0.74 ± 0.05	0.33 ± 0.05	0.70 ± 0.10	8.00 ± 0.07	0.55 ± 0.04	2.33 ± 0.12
Functions	0.57 ± 0.10	2.67 ± 0.05	0.65 ± 0.04	0.33 ± 0.03	0.75 ± 0.10	5.00 ± 0.12	0.50 ± 0.01	2.00 ± 0.10
Recursion	0.49 ± 0.10	2.00 ± 0.10	0.53 ± 0.04	1.00 ± 0.01	0.70 ± 0.03	5.67 ± 0.04	0.55 ± 0.03	3.33 ± 0.02
Searching	0.51 ± 0.11	1.50 ± 0.06	0.62 ± 0.08	0.00 ± 0.08	0.74 ± 0.05	5.67 ± 0.04	0.49 ± 0.06	3.33 ± 0.10
Sorting	0.45 ± 0.03	1.00 ± 0.06	0.60 ± 0.03	0.33 ± 0.07	0.70 ± 0.09	7.67 ± 0.12	0.48 ± 0.07	2.33 ± 0.06
	40-M		L4S		L70b		L8b	
	Avg Succ. Rate	Avg Inter.	Avg Succ. Rate	Avg Inter.	Avg Succ. Rate	Avg Inter.	Avg Succ. Rate	Avg Inter.
Loops	0.48 ± 0.11	2.00 ± 0.02	0.31 ± 0.01	2.81 ± 0.12	0.21 ± 0.05	1.00 ± 0.04	0.10 ± 0.04	1.00 ± 0.03
Conditionals	0.45 ± 0.12	2.00 ± 0.02	0.51 ± 0.11	2.25 ± 0.12	0.32 ± 0.09	3.00 ± 0.02	0.18 ± 0.10	2.00 ± 0.07
Data Struct.	0.44 ± 0.01	1.00 ± 0.01	0.46 ± 0.02	3.46 ± 0.04	0.24 ± 0.04	1.00 ± 0.09	0.11 ± 0.10	0.00 ± 0.01
Algorithms	0.48 ± 0.09	1.00 ± 0.04	0.55 ± 0.12	3.60 ± 0.11	0.33 ± 0.12	1.00 ± 0.09	0.17 ± 0.07	1.00 ± 0.04
Dyn. Prog.	0.34 ± 0.08	0.00 ± 0.10	0.19 ± 0.05	4.21 ± 0.01	0.23 ± 0.03	1.00 ± 0.12	0.08 ± 0.07	1.00 ± 0.06
Error Hand.	0.49 ± 0.05	1.00 ± 0.03	0.42 ± 0.04	2.03 ± 0.06	0.27 ± 0.02	1.50 ± 0.02	0.13 ± 0.07	1.00 ± 0.02
Functions	0.42 ± 0.06	1.50 ± 0.06	0.59 ± 0.10	2.42 ± 0.05	0.24 ± 0.01	2.00 ± 0.12	0.23 ± 0.08	2.00 ± 0.09
Recursion	0.43 ± 0.02	1.00 ± 0.07	0.13 ± 0.02	4.77 ± 0.09	0.29 ± 0.05	1.00 ± 0.11	0.20 ± 0.10	1.00 ± 0.06
Searching	0.47 ± 0.10	1.50 ± 0.04	0.67 ± 0.12	2.32 ± 0.02	0.30 ± 0.01	1.00 ± 0.11	0.20 ± 0.04	1.00 ± 0.05
Sorting	0.39 ± 0.02	0.00 ± 0.04	0.60 ± 0.02	2.28 ± 0.07	0.31 ± 0.05	1.00 ± 0.08	0.13 ± 0.11	1.00 ± 0.06

at "very easy" for 4o and L-405b, respectively). As such, the high number of interventions at lower difficulties are due to smaller sample sizes at these levels combined with specific challenging cases requiring multiple interventions. On the other hand, we can observe that both 4o and L-405b have high intervention rates for challenges related to "conditionals", "data structures", "algorithms", and "dynamic programming". Looking at the distributions of nodes per concept as shown in Figure 6.10a and 6.10b reveals that these concepts also have a high number of nodes in the deeper parts of the tree, meaning that *PrismBench* has identified that these concepts at high complexities have shown to be challenging for the models and has focused on these areas in order to thoroughly analyze models' capabilities.

Since interventions in *PrismBench* are performed by the model itself through the *Problem Fixer*, the combination of success rate and number of interventions effectively measures the model's program repair capabilities. 4o and L-405b demonstrate strong program repair abilities with both high intervention and success rates. For example, at "very easy" difficulty, 4o



(a) Node distribution for 4o

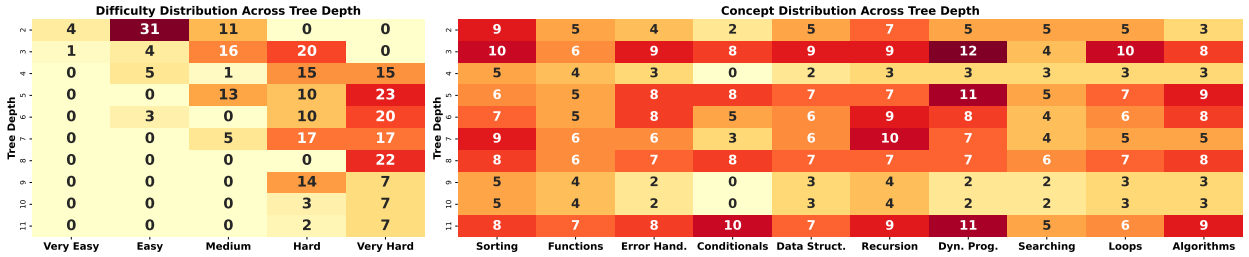


(b) Node distribution for L-405b

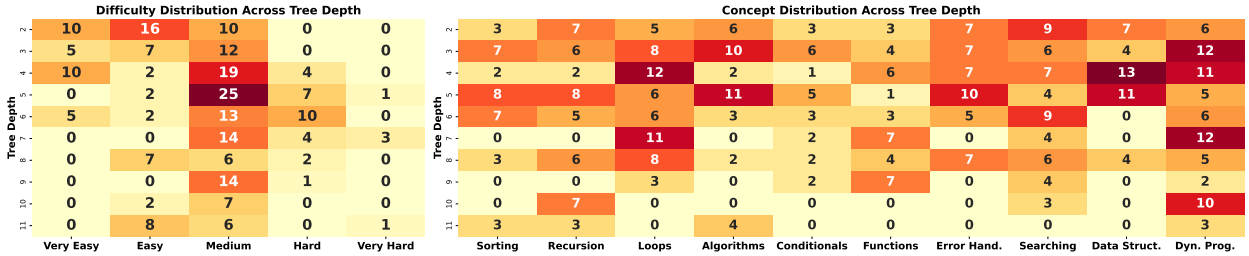
Figure 6.10 Node distributions for 4o and L-405b averaged over 3 independent runs. The numbers in each cell indicate the number of nodes.

shows 4.67 interventions with 0.83 success rate, L-405b shows 3.67 interventions with 0.85 success rate. As such, we can observe that when these models encounter failures, they can effectively analyze their own code, understand test failures, and implement successful fixes. This program repair capability persists even at higher difficulty levels, though with decreasing effectiveness. On the other hand, L-70b and L-8b have a lower number of interventions but significantly lower success rates as well. Furthermore, their success rates remain low despite interventions. For example, L-8b shows minimal interventions across "very easy" and "easy" but maintains very low success rates (0.19 for "very easy", dropping to 0.00 after "easy"). This indicates that even when given full context, including the original solution, test cases, and error outputs, these models struggle to identify and fix problems in their generated code.

GPT-OSS and L4S stand out with the highest intervention rates compared to the other models, with GPT-OSS's solutions requiring the highest intervention rates compared to all the other models, regardless of the concept or difficulty level of the underlying challenge. This indicates that GPT-OSS's first attempt at a solution often fails at first run. However, this also indicates that GPT-OSS has the highest program repair capability: given feedback about the failure, the original solution, and test cases, it reliably identifies root causes and produces a correct solution. Furthermore, when coupled with the node distribution across difficulties and concepts as shown in Figure 6.11a, we can observe that GPT-OSS quickly



(a) Node distribution for GPT-OSS



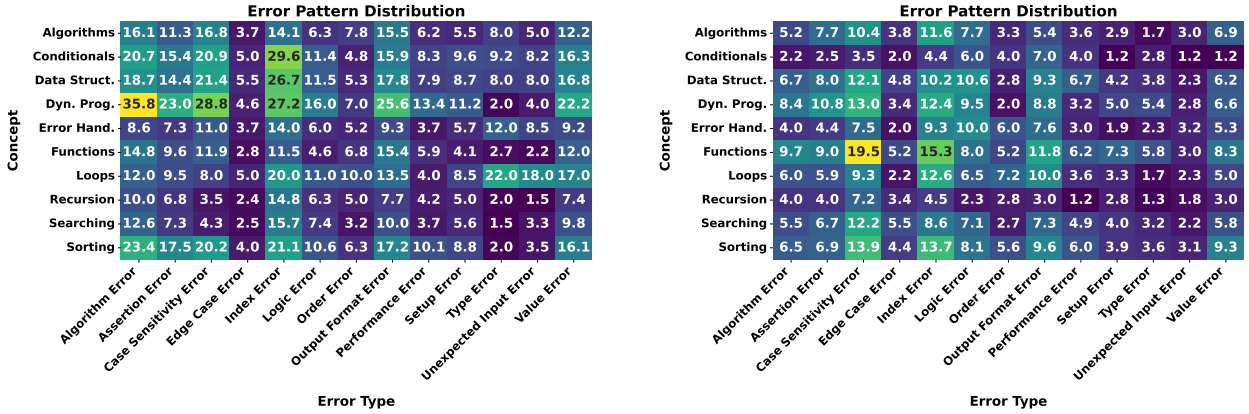
(b) Node distribution for L4S

Figure 6.11 Node distributions for GPT-OSS and L4S averaged over 3 independent runs. The numbers in each cell indicate the number of nodes.

progresses beyond "very easy" to "medium" level challenges and *PrismBench* focuses on challenges with higher levels of difficulty for determining its capability. In contrast, L4S displays inconsistent performance across the search tree. When aggregated by difficulty level, it displays high success rates and moderate to low number of intervention calls, however, as shown in Figure 6.11b, it visits far fewer nodes at "hard" and "very hard" difficulty levels compared to the other top models. We can observe that as *PrismBench* progresses deeper through the tree, once it reaches "hard" difficulty levels, it repeatedly falls back to "medium" and "easy" level challenges as indicated by the node difficulty distributions across depths in Figure 6.11b. However, despite its inability to sustain problem-solving at higher difficulties, from table 6.5, we can observe that, similar to GPT-OSS, it has a high program repair capability.

Figure 6.12 shows the error pattern analysis per concept for 4o and L-405b. The numbers in each cell represent the average occurrence of each error type per concept across 3 independent runs. 4o shows significantly higher errors in algorithm implementations, particularly in challenges related to "dynamic programming" where algorithm implementation errors peak at 35.8 occurrences and errors related to case sensitivity peak at 28.8. Index error rates in generated code for challenges involving "conditional" and "data structure" concepts (29.6 and 26.7 occurrences respectively) further demonstrate 4o's specific struggle with complex

pointer and array manipulations. L-405b's errors, on the other hand, are mainly in "function" implementation challenges (19.5 occurrences for case sensitivity errors, 15.3 for index errors). We can observe that L-405b maintains consistent performance across most tested concepts, with notably lower error rates in recursive implementation challenges (consistently below 4.0 occurrences) compared to 4o. However, similar to 4o, L-405b also struggles with "dynamic programming", "sorting", and "data structure" challenges.



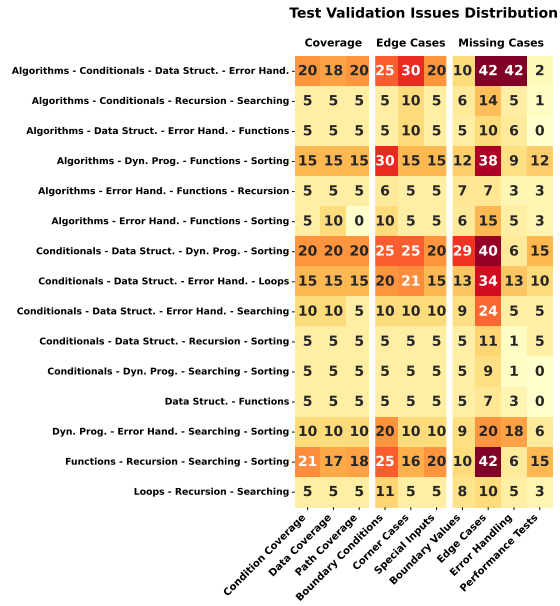
(a) Error pattern distribution for 4o

(b) Error pattern distribution for L-405b

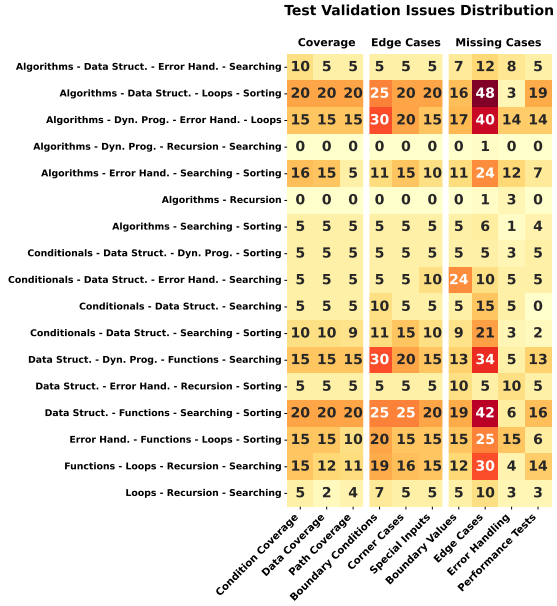
Figure 6.12 Error pattern distributions for 4o and L-405b averaged over 3 runs. The numbers in each cell indicate the number of times each error was raised.

The most encountered error types (algorithm implementation, case sensitivity, and index errors) are consistently related to implementation details rather than fundamental algorithmic understanding. This observation is reinforced by the notably lower frequency of type, setup, and corner case errors across both models and all tested programming concepts. These patterns suggest that while both models demonstrate sound algorithmic understanding, their primary struggle lies in generating the correct code for solutions and tests. Analyzing test validation issues allows us to pinpoint whether the errors stem from incorrectly generated solutions or incorrectly generated tests by the models. Figure 6.13 presents the distribution of test validation issues for both 4o and L-405b across concept combinations. Each cell indicates how often a specific validation problem for a test, such as incorrect condition coverage or incorrect boundary checks, was identified. By comparing these data with the error pattern distributions in Figure 6.12a and 6.12b, we can discover correlations between the root cause of encountered errors and the concept areas where those errors were raised most frequently.

As mentioned in Section 6.5.1, both GPT-OSS and DS3, frequently rely on Python's built-in functionalities (e.g., using Python's in-place `sort` function on arrays) alongside standard and third-party libraries in their solutions (e.g., using `itertools` or `numpy` for array processing),



(a) Test validation issues distribution for 4o

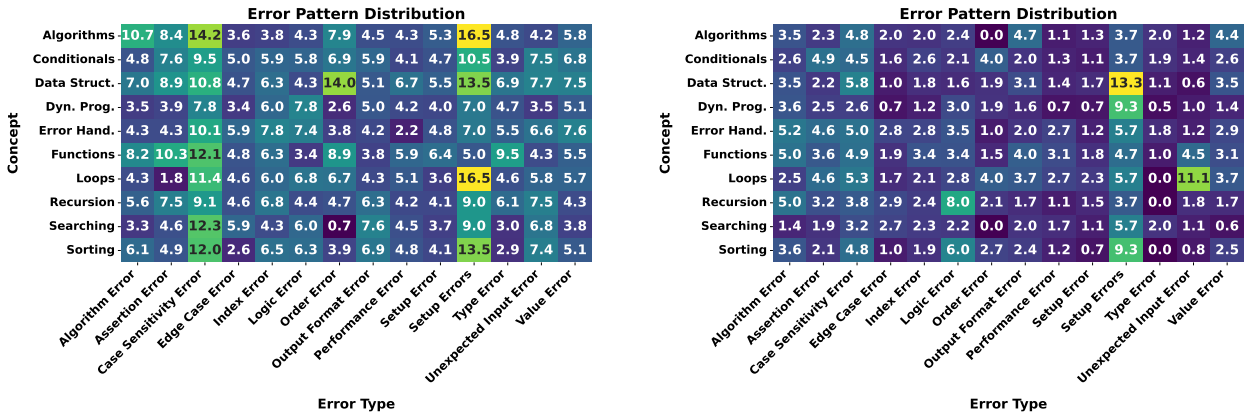


(b) Test validation issues distribution for L-405b

Figure 6.13 Test validation issues distribution for 4o and L-405b averaged over 3 runs. The numbers in each cell indicate the number of times each issue was identified.

particularly in challenges with high difficulty levels or multiple concept combinations. As our aim in *PrismBench* is to evaluate the models' coding capability rather than library knowledge, the *interactive sandbox* provides only a base Python environment without additional packages. Consequently, any dependence on third-party libraries leads to failure regardless of solution logic. Figure 6.14, summarizes the error pattern distributions for GPT-OSS and DS3. Similar to 4o and L-405b failures on the most challenging concepts for the model are mostly caused by algorithm and logic errors; however, both GPT-OSS and DS3 show a high concentration of errors clustered around "setup errors" and "unexpected input errors". These error types are raised when executions fail due to missing imports, incompatible program structure, or library usage outside what is available inside the environment. Moreover, by comparing Figures 6.12 and 6.14, we can observe that GPT-OSS has an overall higher level of error counts compared to 4o, L-405b, and DS3, which is consistent with its higher *Problem Fixer* intervention ratios as shown in Tables 6.4 and 6.5.

For 4o, the combinations of concepts that have the highest number of test validation failures are [algorithms, conditionals, data structures, error handling] and [functions, recursion, searching, sorting]. These concepts also have the highest number of errors, as shown in Figure 6.12a, particularly with index and case sensitivity errors. Furthermore, as shown in



(a) Error pattern distribution for GPT-OSS

(b) Error pattern distribution for DS3

Figure 6.14 Error pattern distributions for GPT-OSS and DS3 averaged over 3 runs. The numbers in each cell indicate the number of times each error was raised.

Figure 6.10a, *PrismBench* has specifically focused on these concepts by generating a high number of nodes for thorough validation and isolation of issues. This indicates that many of 4o's generated tests have the same underlying root cause of its generated solutions (for instance, mishandling pointer or array indices). Moreover, the high frequency of numeric and string value assertions that fail in these tests suggests that 4o often struggles to produce fully consistent test inputs or expected outputs, leading to assertion failures even when the generated solution is correct. We can observe a correlation between test validation issues and error types, demonstrating that these failures are not only in the generated solutions but also in the generated tests. The highest validation issues appear in concepts requiring numeric and string value assertions. This suggests that 4o struggles with processing such concepts during test generation. Therefore, even when 4o produces correct solutions, its limitations in numerical and string processing lead to incorrect test assertions, resulting in failures and error cascades. These issues are also reflected in the success rates and visit counts of concepts as shown in Figure 6.15, where we observe lower success rates and higher visit counts for the combinations of concepts with high error rates and test validation issues.

L-405b on the other hand, consistently struggles with "function" and "sorting" concepts, especially when "data structures" or "searching" are also included as shown in Figure 6.16. The test validation issues in Figure 6.13b demonstrate that combinations like [data structure, function, searching, sorting] exhibit high incorrect coverage issues and a large number of missing or incomplete test cases. Similarly, the error pattern distribution for L-405b in Figure 6.12b shows peaks in case sensitivity and index errors whenever function implementations are tested. We can observe that L-405b exhibits different root causes for failures compared

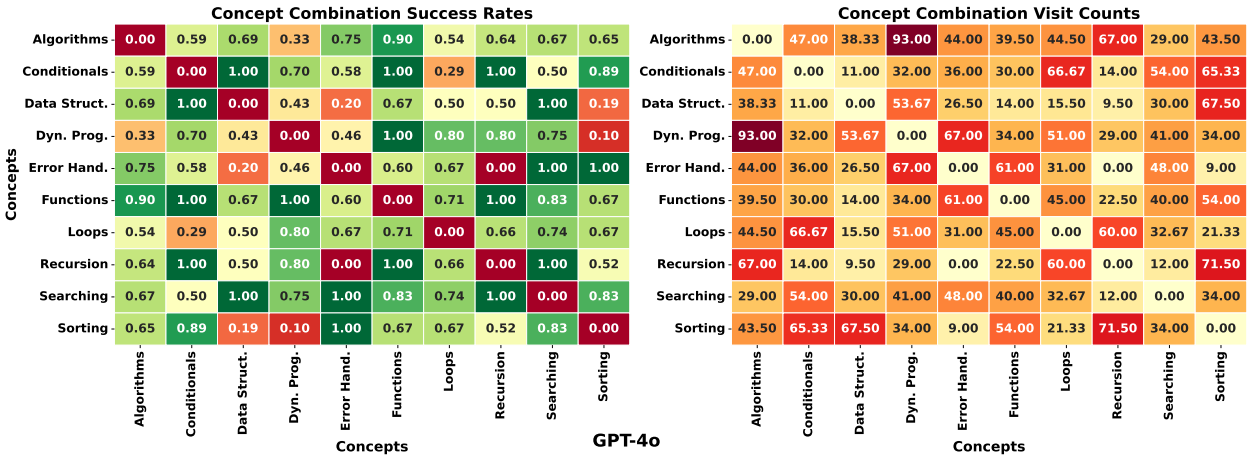


Figure 6.15 Details on the concept combination effects on 4o's performance. The right matrix displays the average success rates for all nodes related to each specific combination. The left matrix displays the average number of times each concept combination was visited in the search tree, regardless of success/failure.

to 4o, particularly in concepts combination of "data structures" and "error handling".

By correlating test validation issues and the solution patterns shown in Figure 6.17, we can see that L-405b's failures primarily stem from syntax errors and hallucinations rather than logical errors as evidenced by the high number of failures in using built-in data types (arrays, list, dictionary, etc.). The lowest-performing nodes and their corresponding patterns show that L-405b frequently generates non-existent syntax (e.g., non-existent built-in function calls, incorrect syntax for using built-in data types, etc.), creating a situation where both the generated code and its corresponding tests are incorrect. This leads to the high intervention rates observed in Table 6.5, as the *Problem Fixer* repeatedly intervenes to correct both solution and test issues.

As shown in Figure 6.18, GPT-OSS's performance remains consistently high across most concept combinations, maintaining high success rates even as challenges become more complex. However, it demonstrates persistent weaknesses in challenges involving "dynamic programming" and "error handling" relative to other concepts. Interestingly, we can observe that nodes which combine "sorting" with "functions", "loops", or "recursion" are frequently revisited across the search tree. This pattern is also present in GPT-OSS's error distributions as displayed in Figure 6.14a, where nodes with these concepts have a high rate of "setup" and "unexpected input" errors. This is a direct result of GPT-OSS's over-reliance on third-party libraries, which are not available in the environment, and its tendency to define nested functions. However, from Figure 6.18, we can observe that these nodes have a high overall success

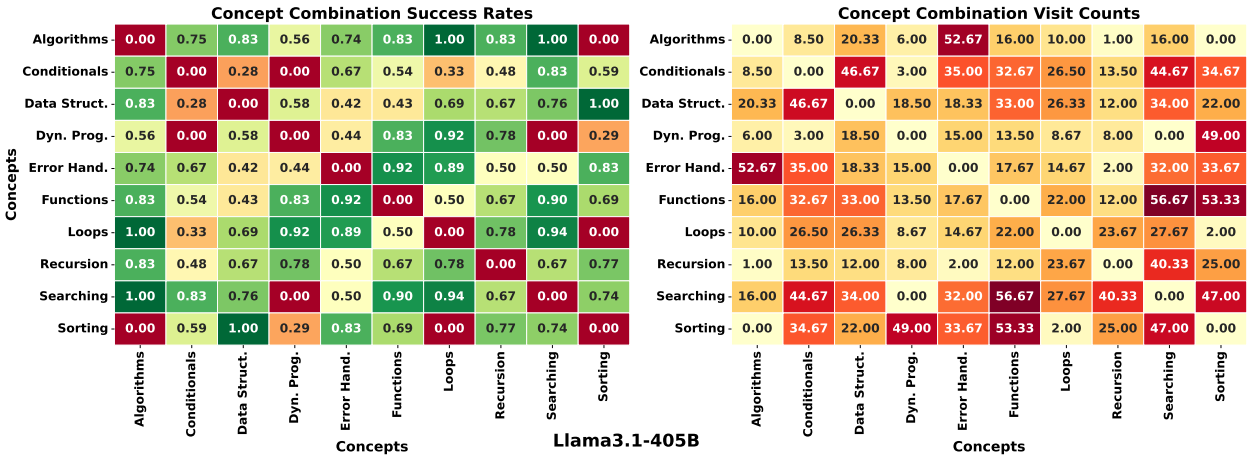


Figure 6.16 Details on the concept combination effects on L-405b's performance. The right matrix displays the average success rates for all nodes related to each specific combination. The left matrix displays the average number of times each concept combination was visited in the search tree, regardless of success/failure.

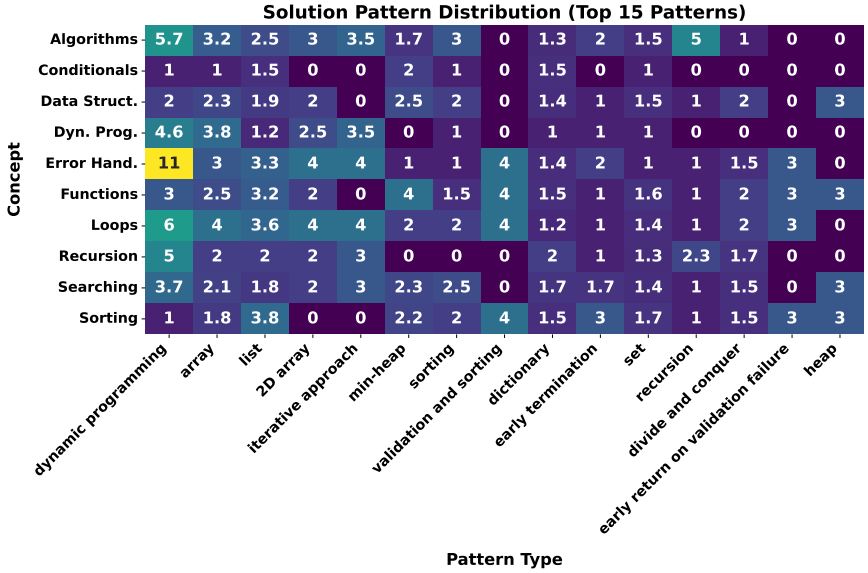


Figure 6.17 Patterns identified in solutions distribution for L-405b averaged over 3 runs. The numbers in each cell indicate the number of times each pattern was identified.

rate. Investigating the solutions for these nodes reveals that GPT-OSS's first attempts at solving challenges requiring array manipulation rely on third-party libraries instead of built-in standard libraries and functionalities. It is only during the subsequent attempts, where the feedback contains information about these libraries not being available, that it generates a correct solution. As such, GPT-OSS exhibits a different coding profile compared to the

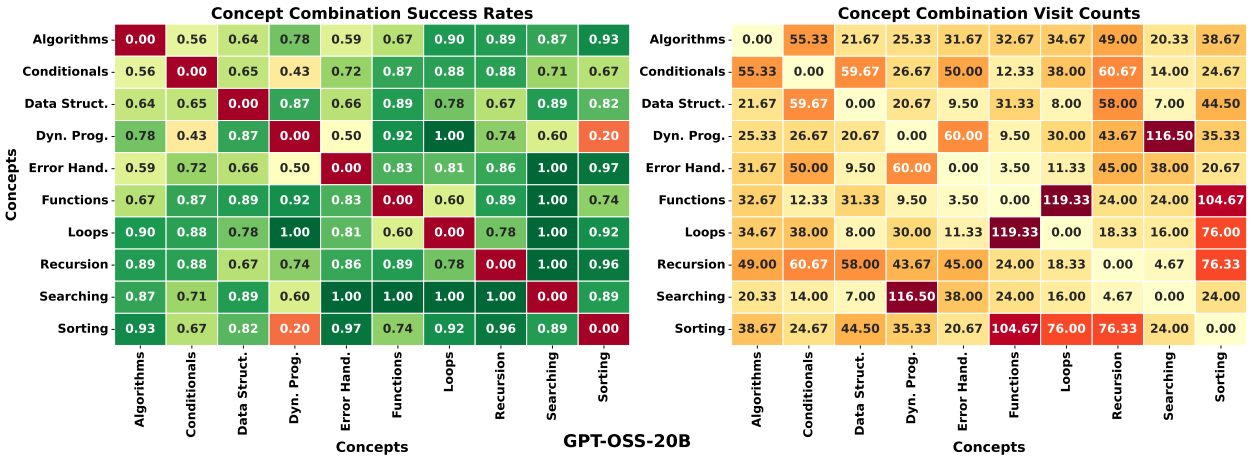


Figure 6.18 Details on the concept combination effects on GPT-OSS's performance. The right matrix displays the average success rates for all nodes related to each specific combination. The left matrix displays the average number of times each concept combination was visited in the search tree, regardless of success/failure.

much larger 4o and L-405b, which use Python's standard libraries.

It is important to note that these insights come from *PrismBench*'s automated analysis. The search trees generated by the framework enable deeper investigation of behavioral patterns and contain detailed analysis for each node. We only highlight the most significant behavioral patterns observed.

6.5.3 Effects of Scale

Given the evaluation results on the GPT-4o and Llama3.1 families, in this section, we investigate the effects of model scale on code generation and problem-solving capabilities. Importantly, we do not include GPT-OSS and L4S in our analysis in this section, as both models have different architectures compared to the others.

GPT-4o

GPT-4o and GPT-4o Mini, developed by OpenAI, are part of the same model family but differ in scale, performance, and application focus [315]. GPT-4o is the high-performance, multimodal flagship model optimized for complex tasks requiring deep reasoning and nuanced language understanding, while GPT-4o Mini is a lightweight, cost-efficient variant designed for speed and accessibility, prioritizing rapid token generation and affordability. While both models share core architectural features like Transformer-based design and multimodal ca-

pabilities, GPT-4o Mini is reported to be significantly smaller than GPT-4o.

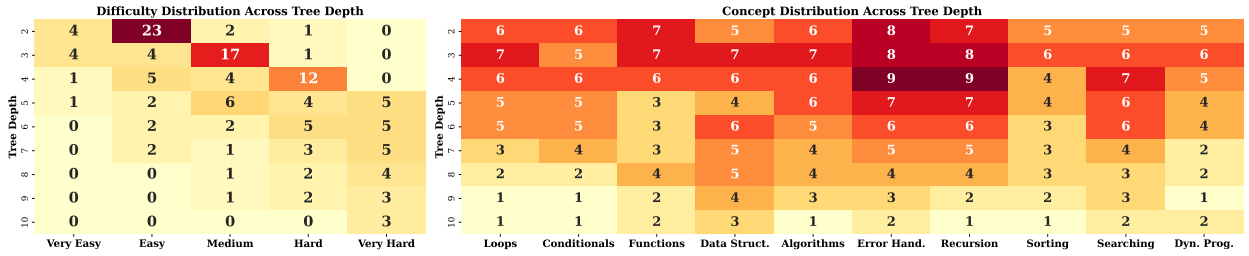


Figure 6.19 Node distribution for 4o-M averaged over 3 runs. The numbers in each cell indicate the number of nodes.

Figure 6.19 displays the node distribution and visit counts of 4o-M throughout the search tree. In the previous sections, we presented performance results for 4o, with its corresponding node distributions presented in Figure 6.10a. Comparing the distributions of 4o with 4o-M shows us how scale impacts performance. While the majority of 4o's nodes are distributed in challenges with "hard/very hard" difficulty and deeper parts of the tree (as shown in Figure 6.10a), we can observe that for 4o-M, the majority of nodes are distributed between challenges with "medium" and "hard" difficulty and in shallower depths. This is also evident in the search tree for 4o-M as shown in Figure 6.26.

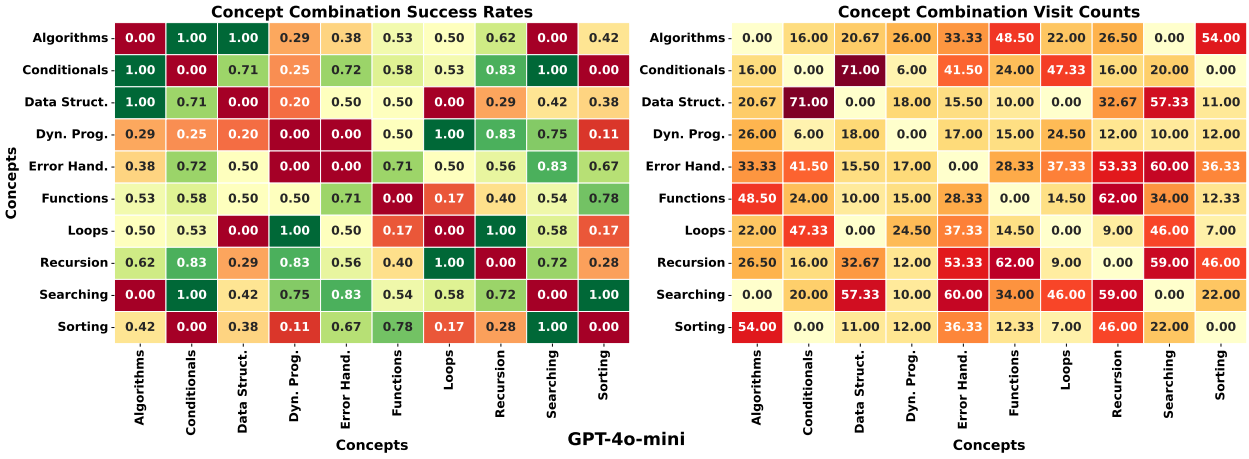


Figure 6.20 Details on the concept combination effects on 4o-M's performance. The right matrix displays the average success rates for all nodes related to each specific combination. The left matrix displays the average number of times each concept combination was visited in the search tree, regardless of success/failure.

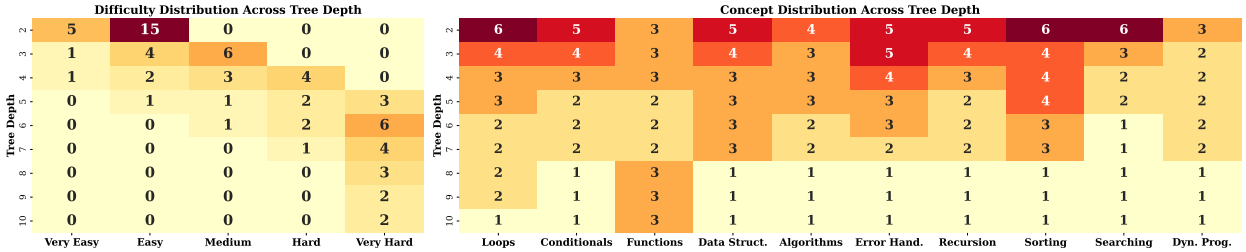
Figure 6.20 shows the success rates and visit ratios for nodes corresponding to different concept combinations. As discussed in Section 6.5.1, we can see that the majority of 4o-M's

failures occur when it encounters combinations of concepts that require compositional reasoning. For instance, we can observe that 4o-M has relatively low success rates for "dynamic programming", which fall even lower when the challenge combines another concept with "dynamic programming".

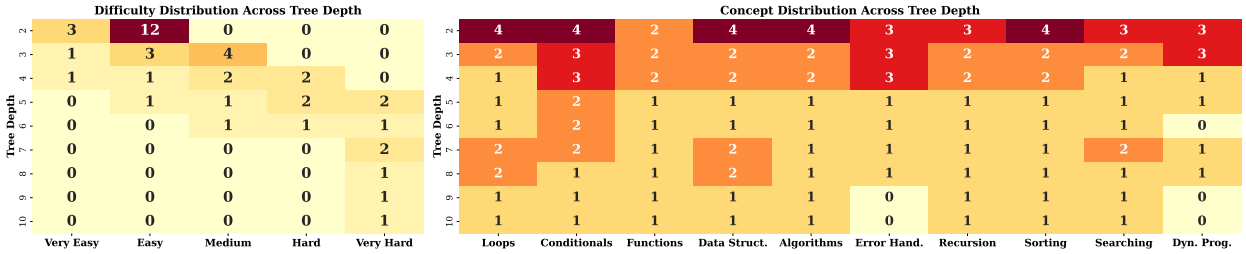
Llama 3

The Llama 3 herd of models, developed by Meta, is a family of LLMs designed to support multimodality, coding, reasoning, and tool use. The term "herd of models" refers to the diverse range of models within the Llama 3 family, each tailored for specific applications [317]. The flagship model, L-405b, is a dense Transformer architecture with 405 billion parameters and a context window of up to 128,000 tokens, enabling it to handle extensive datasets and complex tasks. While these models share foundational training data and post-training processes, they differ in architectural scale—such as the number of layers, model dimensions, attention heads, and FFN dimensions—to optimize performance across varying use cases. This allows us to leverage *PrismBench* to systematically evaluate how architectural and parametric scale impact code-generation capabilities. While we have already presented performance results for L-405b (the most capable variant) in prior sections, this section focuses on analyzing performance differences across scaled-down versions of the Llama 3 family.

As shown in Figure 6.21a and 6.21b, the depth of explored nodes drops sharply for smaller models, indicating limitations in handling more difficult problems. In particular, the node distribution for L-8b shows that only the shallow parts of the tree (depths 2 and 3) and "easy" challenges have been explored in depth. The inability to explore deeper nodes suggests that the model fails to generate correct solutions when challenges are more difficult or contain multiple programming concepts. Conversely, L-70b reaches deeper parts of the tree more often and is capable of reaching nodes with "very hard" difficulty to an extent (even though it fails at all of them) as shown in Table 6.4. The node distributions of the search trees generated for L-405b, L-70b, and L-8b) clearly demonstrate how models' scale plays an important role in their problem-solving and code-generation capabilities. The search trees themselves for these two models (L-70b and L-8b), as presented in Figure 6.30, further show how these models struggle to complete challenges as they become more difficult. Figure 6.30b shows how the majority of nodes for L-8b are generated in Phase 3 (highlighted in blue). As explained in Section 6.3.4, Phase 3 is responsible for comprehensively inspecting areas of failure, and as such, we can see that L-8b consistently fails with high failure rates, with the majority of nodes being generated in Phase 3. On the other hand, L-70b shows a slightly better performance as pictured in Figure 6.30a. In the same manner as L-8b, the majority



(a) Node distribution for L-70b



(b) Node distribution for L-8b

Figure 6.21 Node distributions for L-70b and L-8b averaged over 3 runs. The numbers in each cell indicate the number of nodes.

of L-70b's nodes are generated during Phase 3, However, we can see that unlike L-8b, many nodes were also generated in Phase 2, indicating that the model was capable of solving some of these challenges albeit with low success rates.

Figure 6.22 and 6.23 further demonstrate performance degradation as the models get smaller. The success rates of concept combinations decrease significantly for L-8b, particularly for tasks requiring more advanced strategies (e.g., "dynamic programming" or multiple nested constructs). In comparison, L-70b shows moderate success with simpler "loops" and "conditionals" but similarly struggles to sustain the performance under combined, higher-level concepts.

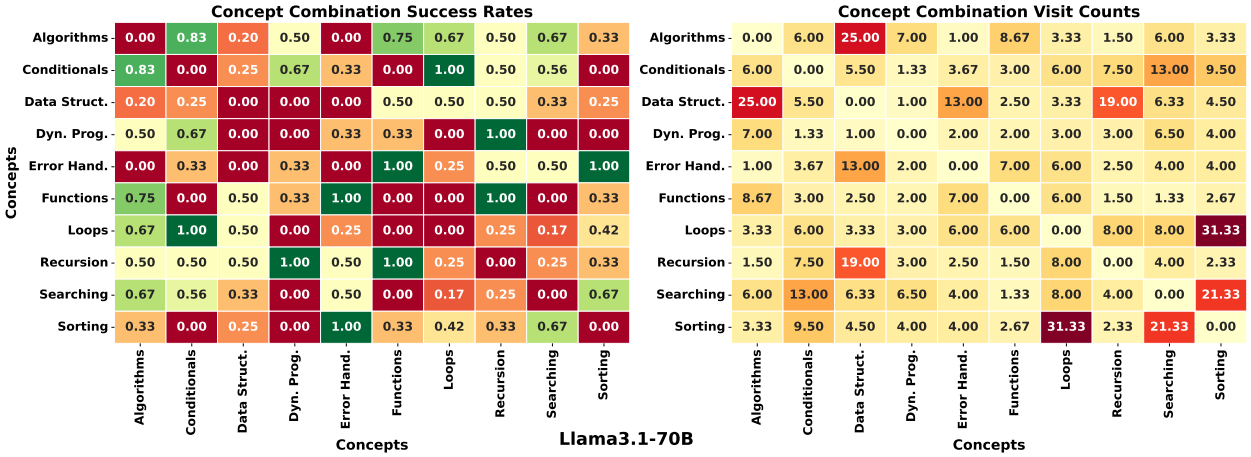


Figure 6.22 Details on the concept combination effects on L-70b's performance. The right matrix displays the average success rates for all nodes related to each specific combination. The left matrix displays the average number of times each concept combination was visited in the search tree, regardless of success/failure.

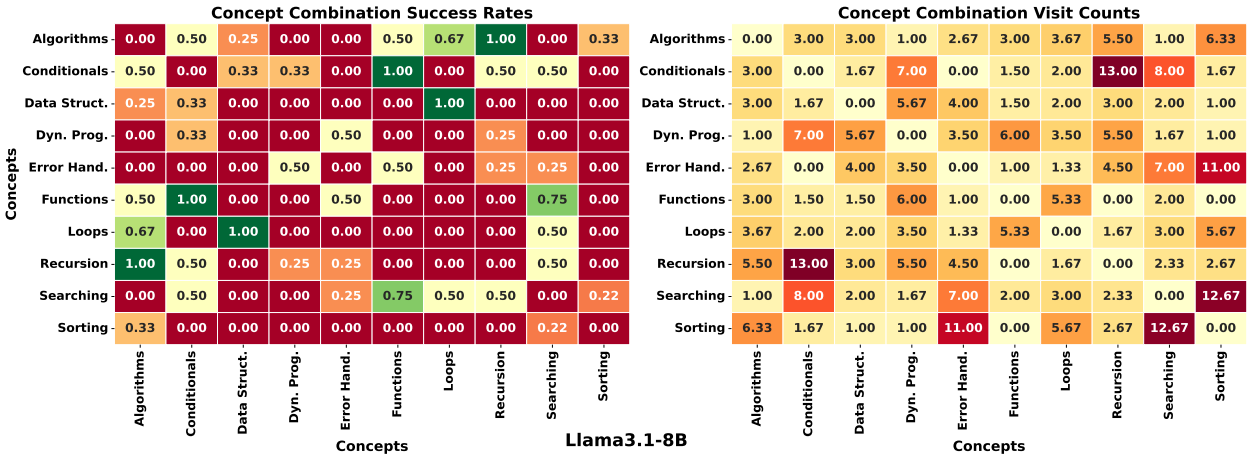


Figure 6.23 Details on the concept combination effects on L-8b's performance. The right matrix displays the average success rates for all nodes related to each specific combination. The left matrix displays the average number of times each concept combination was visited in the search tree, regardless of success/failure.

6.5.4 Sample Trees

This section presents the minimized versions of the search trees generated for all models examined in this study. To ensure brevity, the trees presented here focus solely on the concepts, difficulty levels, and scores of each node, along with the phase during which they were generated. The original trees, however, are much more detailed, but they would not fit in the content of this paper. The original trees contain the complete challenge description, the generated solutions, tests, attempts, and the corresponding analysis done at each node. This information allows for a comprehensive and fine-grained evaluation of model behavior at each node in the search tree. We have included the full original trees in our replication package [308]. The nodes for each phase are color-coded for distinction, with yellow nodes representing those generated in Phase 1, green nodes representing those generated in Phase 2, and blue nodes representing those generated in Phase 3. The edges indicate parent-child relationships, with red edges indicating a significant decrease in the child node's TD value compared to its parent, and green edges indicating otherwise. Each node is associated with specific attributes: the concepts related to the node are under "Concepts," the difficulty level under "Difficulty," and the number of times the node has been visited, under "Visits."

6.6 Threats To Validity

Threats to internal validity concern factors internal to our work that could have influenced our study. The inherent stochasticity of LLMs which results in variability of results across runs is the most important threat to our work's internal validity. we aimed to address this threat at each step of the process through using TD(0) scoring for nodes, ϵ-greedy policies for node selection, and node value convergence checks at each evaluation phase as detailed in Sections 6.3.2 and 6.3.4. Additionally, while our multi-phase search strategy is designed to focus on promising areas of the search space, the stochasticity in LLM's performance may introduce state selection bias by underexploring edge cases. We mitigate this threat by conducting multiple trials for each model and reporting averaged results across trials to account for variability and reduce the impact of outlier behaviors.

Threats to construct validity concern the relationship between theory and observation. In the context of our work, the main construct threat is whether our proposed evaluation metrics (as we detail in Section 6.3.6) holistically capture the multiple aspects of LLMs' code generation capabilities. We mitigate this threat by executing generated code within an isolated sandbox environment, ensuring that our evaluation is grounded in the actual result of the models' outputs. Furthermore, as detailed in Section 6.3.3, we isolate the different stages of code generation, namely solution generation, test generation, and program repair, by using separate agents for each step. The other threat to our work's construct validity is that Phase 3's pattern analyses rely on a judge model (4o in this study) and while the judge model can be changed to any available model, there still exists the risk of the judge's capability ceiling and potential bias resulting in incorrect analysis results. We address this threat by restricting the judge model's role to post hoc analysis so that it does not influence the overall search and benchmarking process.

Threats to external validity concern the generalizability of our findings beyond the specific setup used in this study. Our evaluations were carried out on eight LLMs and a finite set of concepts and difficulty levels, which, although diverse, may not capture the full range of coding tasks encountered in practice. Additionally, as the benchmarking process is computationally and financially expensive, each node's evaluation requires multiple LLM calls via API or local models (as detailed in Sections 6.3.3 and 6.4.4), we limited the number of tasks and the depth of tree expansion at each phase (Section 6.4.3). These constraints may impact the extent to which our results generalize to larger-scale or different task distributions.

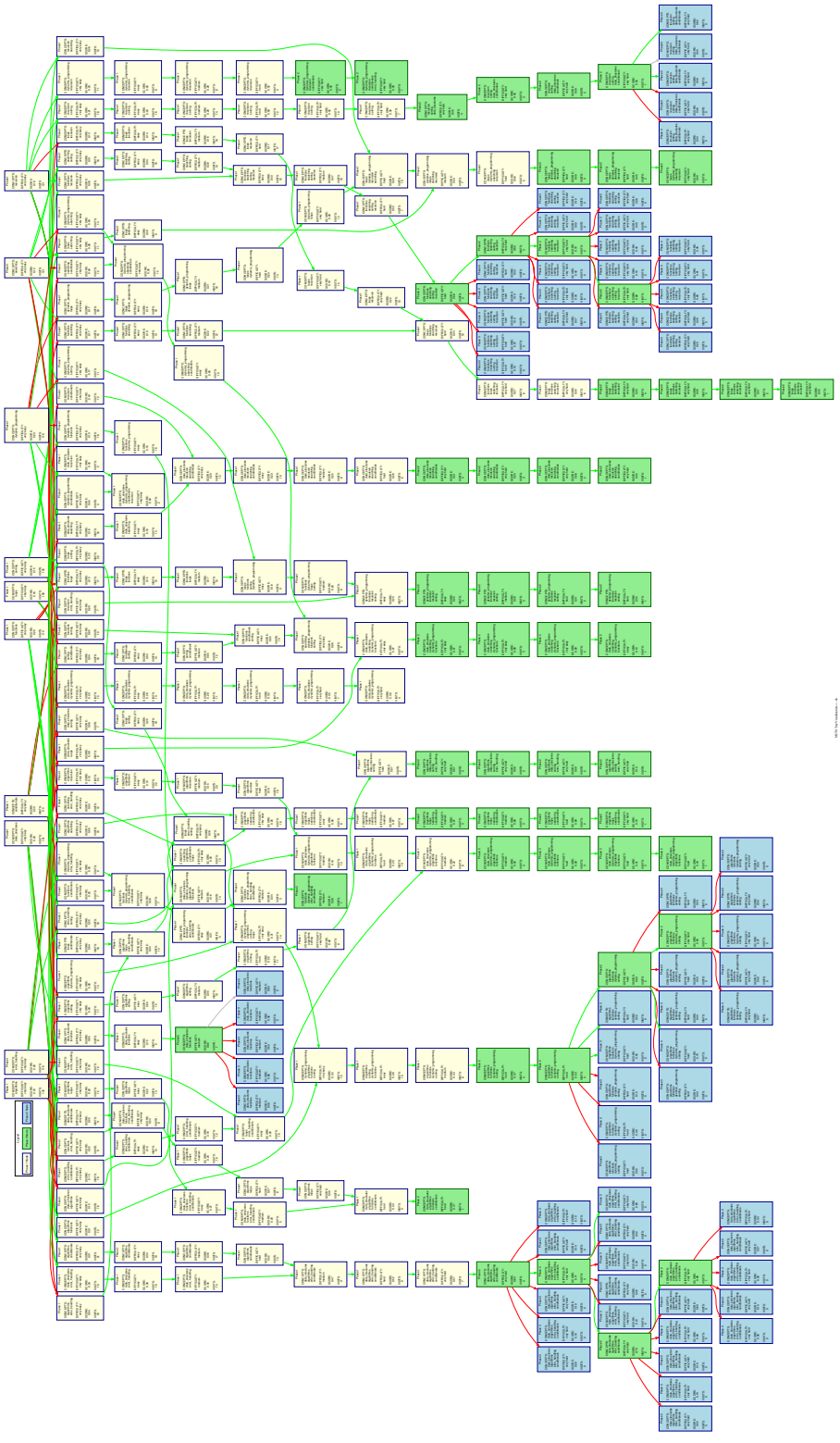


Figure 6.24 Search tree generated for 4o

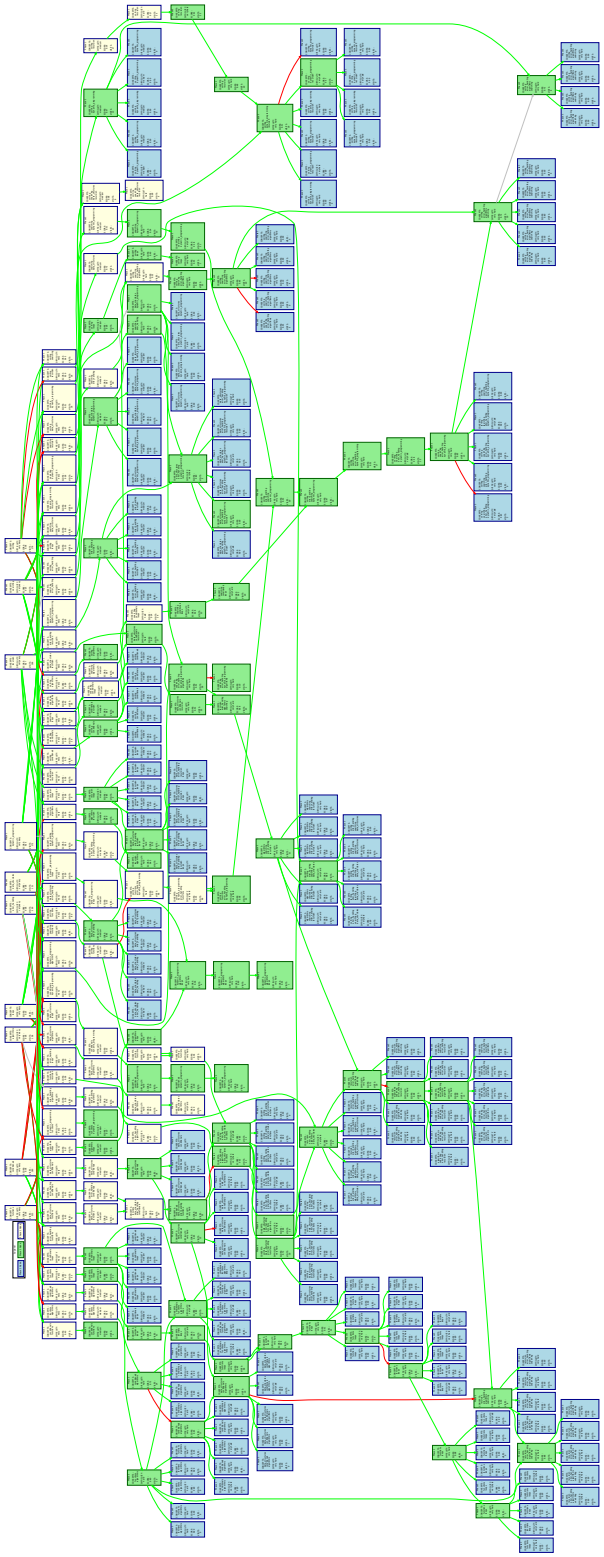


Figure 6.25 Search tree generated for GPT-OSS

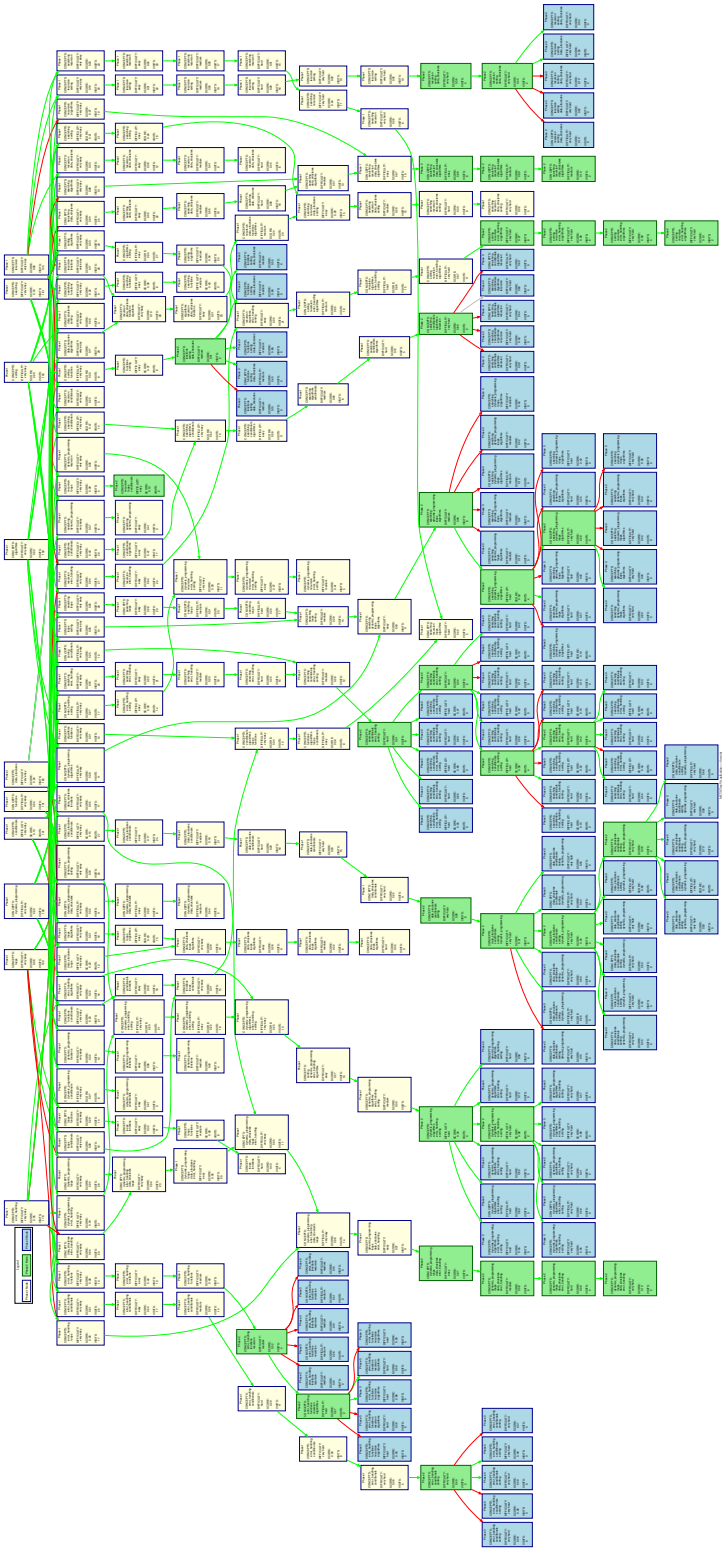


Figure 6.26 Search tree generated for 4o-M

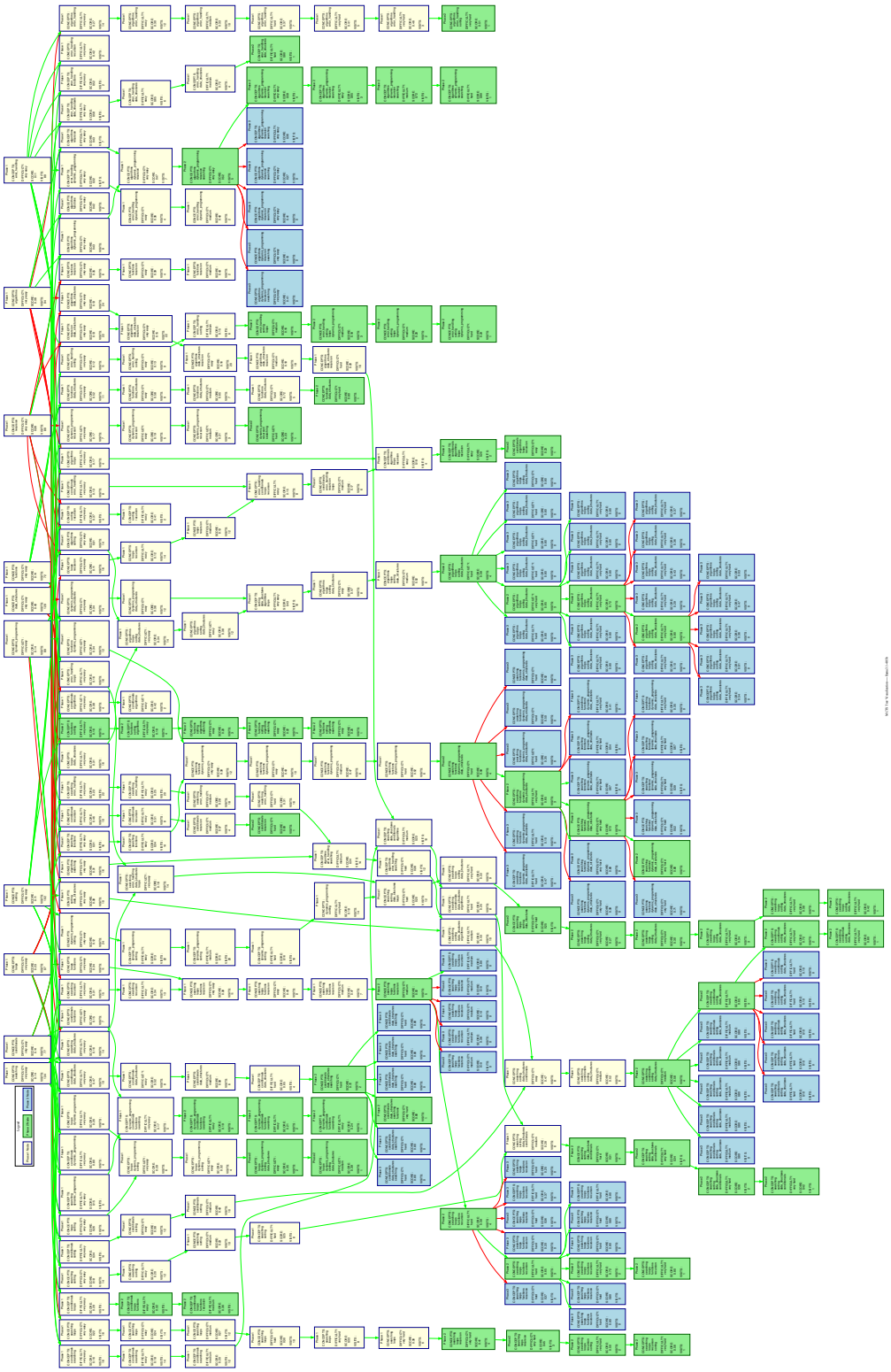


Figure 6.27 Search tree generated for L-405b

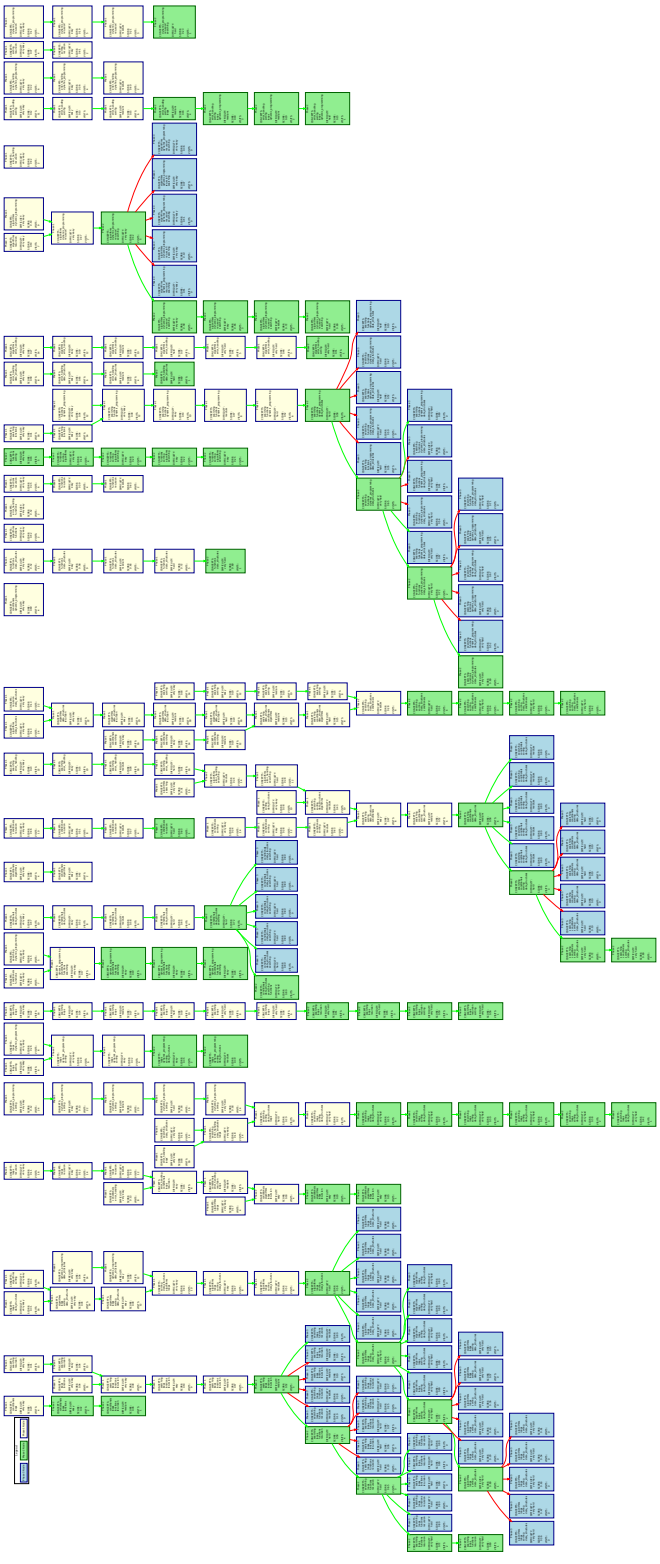


Figure 6.28 Search tree generated for DS3

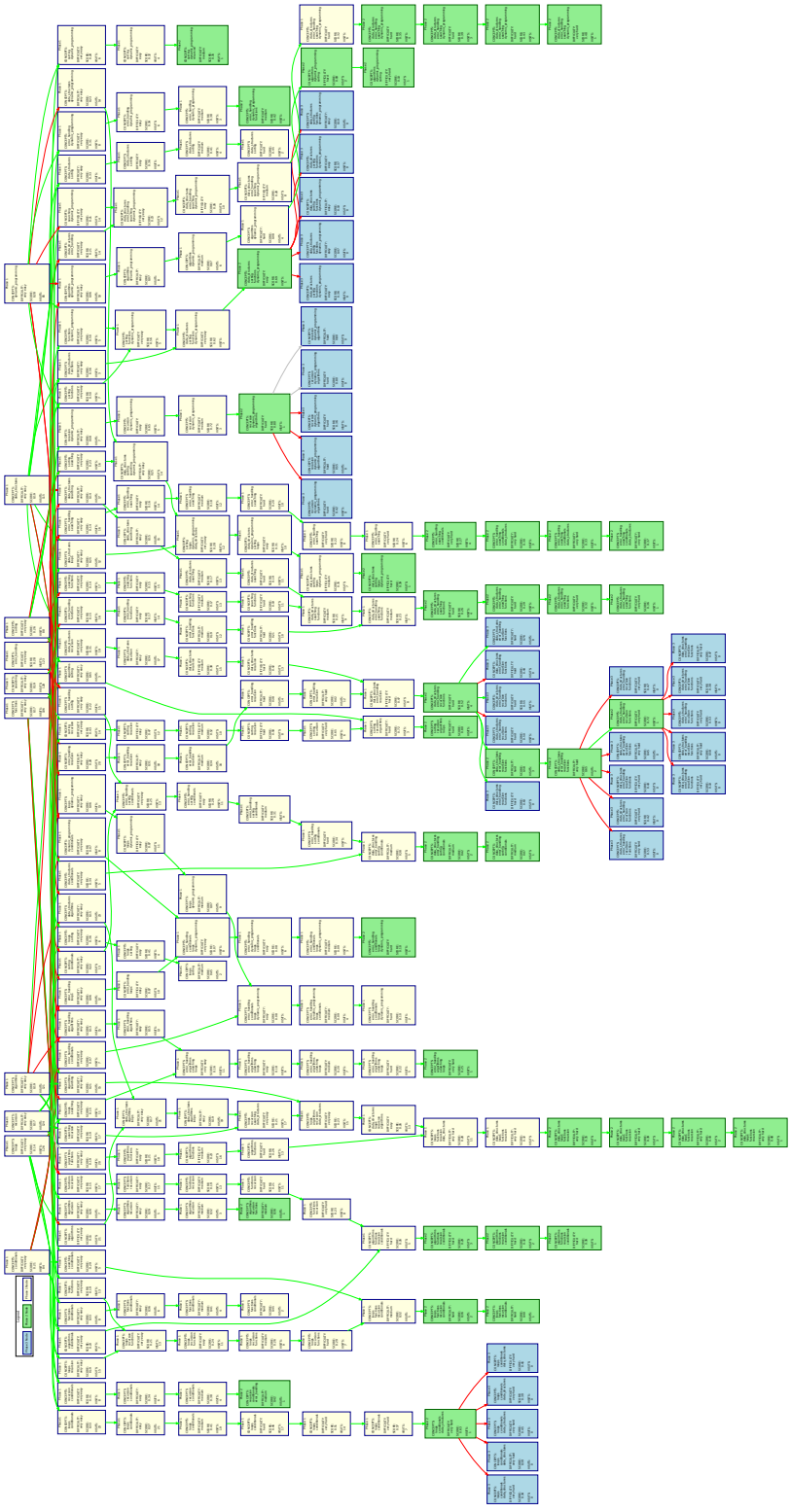


Figure 6.29 Search tree generated for L4S

(a) Search tree generated for L-70b (b) Search tree generated for L-8b

Figure 6.30 Search trees generated for L-70b (a) and L-8b (b)

6.7 Chapter Summary

In this chapter, we introduced *PrismBench*, a dynamic benchmarking framework that models the evaluation space as a search tree and uses MCTS to systematically explore evaluation scenarios. Unlike prior approaches, *PrismBench* dynamically analyzes how LLMs approach problems, adapt to feedback, and handle increasing complexity using a structured multi-phase pipeline and specialized agents to uncover systematic model weaknesses. Our key contributions are as follows:

- We formalized the space of code generation tasks as a search tree over programming concepts and difficulty levels, allowing for adaptive and targeted evaluation of LLM capabilities.
- We introduce a multi-phase evaluation strategy: (1) Capability Mapping to assess baseline strengths, (2) Challenge Discovery to identify systematic weaknesses, and (3) Comprehensive Analysis to investigate failure patterns and root causes.
- We proposed a set of metrics in order to provide fine-grained insights into model behavior beyond standard pass/fail metrics.
- We conduct experiments across eight state-of-the-art LLMs and show that while larger models demonstrate stronger general capabilities, they still struggle with compositional reasoning, high-difficulty tasks, and consistent test generation.

Additionally, we provide *PrismBench* as an open-source, extensible framework [308] for conducting such evaluations to allow practitioners to conduct trustworthy, targeted benchmarking strategies as LLM capabilities continue to evolve.

CHAPTER 7 CONCLUSION

This thesis presents our work for addressing the following high-level research question:

> How can we enhance the transparency and trustworthiness of ML models trained on code for SE tasks?

To address this question, we focused on three components of the ML training and deployment pipeline, namely, model training, auditing, and evaluation. To address each aspect, we defined the following RQs:

RQa: How can we inspect what ML models trained on code learn from their training data?

We addressed this RQ in Chapter 4 and introduced our probing framework *DeepCodeProbe*. *DeepCodeProbe* transforms structured representations of code used to train ML models (e.g., ASTs, CFGs) into a high-level, compact $\langle d, c, u \rangle$ tuple representation that models node order, edge connections, and node labels. This representation is then used for training a lightweight probe in order to recover these tuples from an ML model's internal embeddings to investigate the model's syntax learning and retention capabilities. We validate the entire pipeline at each step (AST/CFG to $\langle d, c, u \rangle$ conversion, $\langle d, c, u \rangle$ tuples encoding syntactic qualities, and internal embeddings validity) to ensure that both the constructed $\langle d, c, u \rangle$ tuples and the extracted embeddings are syntactically valid. By focusing on structured representations of code rather than raw text tokens, *DeepCodeProbe* allows for probing small to mid-size ML models (instead of models with billions of parameters), and addressing the gap left by prior LLM-oriented approaches.

To show *DeepCodeProbe*'s effectiveness, we retrained and probed eight representative models spanning five SE tasks (clone detection, code summarization, automated program repair, code representation learning, and type inference). For each model, we extracted per-layer embeddings, constructed a task-specific probing dataset, and ran a grid search over probe sizes (i.e., number of parameters) in order to ensure fair comparison. Using our proposed probing approach, we studied the effects of data representation quality (whether the provided data transformation pipelines encode syntactic information to be used for training models) alongside the effect of representation choices (whether training the models on the resulting representations results in models retaining syntactic information) by quantifying how accu-

rately the extracted embeddings from each model can be used to reconstruct the $\langle d, c, u \rangle$ tuples and measuring cosine similarities between clone/non-clone pairs to confirm that these embeddings encoded programming language's syntax rather than noise.

Our results demonstrated a clear pattern: models that are trained on explicit structured representations internalize richer syntax. For example, Recoder trained on full ASTs for APR, recovered every tuple component with $\geq 73\%$ accuracy, while text-token baselines such as SummarizationTF stayed below 15%. On the other hand, based on the downstream task, Graph-based FuncGNN excelled at CFG edge information (98% c-accuracy) yet struggled with node order and labels, demonstrating that representation and architecture jointly shape what is learned. Our scaling experiments further revealed diminishing returns: after a moderate size increase, most models plateaued or over-fitted, whereas selectively enlarging layers that exploit structure offered small but reliable gains.

By using *DeepCodeProbe*, we demonstrated that representation quality outweighs sheer parameter count for ML models trained on code. Structured inputs (AST/CFG) enable even lightweight architectures to capture task-relevant abstractions, and probing should be integrated into evaluation pipelines to catch superficial learning early. Based on our empirical analysis we then provided a set of guidelines for practitioners: (1) favor syntactic representations over plain text, (2) scale only the layers that leverage structure as opposed to the entire model, and (3) deploy probing tools before deployment to ensure models encode the syntax they need to act reliably.

Future works can build upon *DeepCodeProbe* in order to explore the generalizability of learned code representations across both tasks and programming languages. Specifically, investigating whether representations trained for a particular SE task (e.g., CCD, APR, code summarization, etc.) can transfer to other tasks by building and comparing the $\langle d, c, u \rangle$ tuples extracted from the same model trained for different downstream tasks. Similarly, probing models trained on one programming language and evaluating their representation quality on another programming language could reveal how well structural representations like ASTs and CFGs generalize across different syntactic paradigms. These studies would help clarify the limits and strengths of structural representations in building broadly trustworthy and lightweight ML models for SE tasks.

RQb: How can we audit LLMs trained on code to ensure data provenance without access to model internals?

We addressed this RQ in Chapter 5 and introduced our framework for dataset inclusion detection, *TraWiC*. *TraWiC* is model-agnostic and frames inclusion detection and data provenance

auditing as a targeted membership inference attack. We parse each source file to extract six elements unique to code: variable, function, and class names (syntactic elements) alongside strings, statement-level comments, and docstrings (semantic elements). Afterwards, we mask each element at a time and query the LLM with a fill-in-the-middle prompt. Using the LLM outputs, we count the exact matches on syntactic elements and partial matches (edit distance) on semantic ones, and use a lightweight classifier (e.g., random forest, SVM, XGB) to detect inclusion based solely on LLM outputs. As such, *TraWiC*'s entire pipeline works without access to the model weights or the original training dataset and only requires a black-box access level to the LLM, making it suitable for auditing LLMs in real-world scenarios.

TraWiC's detection pipeline is carried out through four stages: Extracting and masking elements and then constructing (prefix, suffix) pairs for each extracted element, querying the LLM to predict the masked tokens, counting the number of exact and partial match hits from the LLM outputs to create a per-type hit vector, and detecting dataset inclusion based on the vectors using the classifier. This design keeps the approach interpretable, generalizes across LLM families, and is far less resource-intensive than traditional pair-wise CCD approaches that are used for auditing data provenance.

We evaluated *TraWiC* on a ground-truth corpus of 10,700 source files (across 314 repositories) for three LLMs: SantaCoder alongside fine-tuned versions of LLaMA-2 and Mistral. We show that for detecting single-script inclusion, our approach reached 83.87% accuracy/F-score for SantaCoder and $\geq$ 80% for the other models. Meanwhile, traditional CCD baselines, NiCad, and JPlag were capable of achieving only 47.64% and 55% accuracy, respectively, making them lag far behind *TraWiC* in terms of performance while requiring much more compute at the same time. Furthermore, we conducted comprehensive sensitivity analyses by applying varying levels of identifier obfuscation to the dataset and show that at low to medium obfuscation, *TraWiC*'s F-score drops by only 10%. Finally, we provide an in-depth analysis of the effects of each level of obfuscation across syntactic, semantic, and their combination on performance and feature importance of the inclusion detection pipeline.

Future works can build upon *TraWiC* in order to investigate what other aspects of code can be used for conducting MIAs on a code model. Additionally, a model that can detect dataset inclusion based on the outputs of a code model by comparing the AST representation of the generated output to the AST representation of the input can be trained using RL. Doing so would considerably lower *TraWiC*'s performance bottleneck and allow for constructing an end-to-end solution for dataset inclusion detection that is more robust against code obfuscation.

RQc: How can we evaluate LLMs on end-to-end code generation tasks and determine their capability ceilings?

We addressed this RQ in Chapter 6 by introducing *PrismBench*, a dynamic, multi-agent benchmarking framework to systematically evaluate LLMs' code generation capabilities. In contrast to static benchmarks that rely on fixed datasets and simple pass/fail metrics, with *PrismBench*, we model LLM evaluation as an MDP over a structured tree of programming challenges, with each node in the tree representing a unique combination of programming concepts and difficulty level. Given this formulation, we use a modified MCTS algorithm to explore and traverse this tree, which allows *PrismBench* to dynamically adapt evaluation tasks based on model performance and surface subtle weaknesses and systematic failure patterns in LLMs' code generation.

In order to comprehensively evaluate each aspect of code generation (e.g., solution generation, test suite creation, and program repair), *PrismBench* uses a multi-agent pipeline where each agent handles a specific task and is isolated from the other agents in order to prevent data contamination between agents. This setup ensures reproducible, feedback-driven evaluation signals based on functional correctness rather than textual similarity. We break the evaluation process into three phases: capability mapping (gauging the model's baseline performance), challenge discovery (targeting low-performing areas), and comprehensive analysis (probing root causes of failure through challenge variations). Additionally, in order to allow for fair comparison between LLMs of different capability levels in a dynamic evaluation process, we propose structural, mastery, performance, and diagnostic metrics that go beyond pass/fail metrics such as accuracy or pass@k and capture how models solve tasks, where they struggle, and root causes behind their failures.

To show *PrismBench*'s effectiveness, we benchmarked eight state-of-the-art LLMs on their code generation, test suite creation, and program repair capabilities. Our benchmarking results show that larger models such as 4o and L-405b consistently solved more complex and difficult challenges, demonstrating strong baseline proficiency across a wide range of programming tasks. In contrast, smaller models like L-8b failed on relatively simple challenges, demonstrating limited capacity for general code generation. We show that despite their scale, even the top-performing models struggled with tasks that required compositional reasoning or deep algorithmic understanding, such as dynamic programming or recursion, with 4o-M frequently failing on challenges involving nested logic or the integration of multiple concepts. Furthermore, our evaluations provide insights into model-specific failure patterns: for example, L-405b often produced logically correct solutions that failed due to syntax errors or misunderstanding of task instructions. Finally, we provide an in-depth analysis of the code

generation performance of each of the LLMs under study throughout the benchmarking process and demonstrate how *PrismBench*'s dynamic evaluation pipeline exposes nuanced and systematic weaknesses in how LLMs attempt to solve programming tasks.

7.1 Limitations

In this thesis, we aimed to address three major components of the ML training and deployment pipeline for SE tasks: model training, auditing, and evaluation. While our works contribute meaningfully to these aspects, the broader lifecycle of ML systems contains additional components that were not studied in this thesis. For instance, the process of collecting and cleaning training datasets used for training a model can significantly influence its performance and learned representations. Although we attempted to control for these aspects in *DeepCodeProbe*, we did not do so for *TraWiC* and *PrismBench* as we do not re-train the LLMs under study from scratch on our own curated dataset. Furthermore, the problem of concept drift, in which a model's behavior and performance change over time as it is retrained on new data, is a critical challenge in real-world deployment of ML models and was not explored in this thesis. Beyond these general concerns, several specific limitations could have affected our findings:

- *Model diversity:* while all the frameworks proposed in this thesis are model agnostic (i.e., our frameworks are designed to work with any ML model that falls under the purview of being trained on code), our experimental results and subsequent analyses of these results can benefit from applying our frameworks on more models. For example, while for *DeepCodeProbe* we conduct our study on multiple models across different downstream tasks and architectures, there still exists a plethora of model architectures and data representation approaches used for training small to mid-size models on code. Similarly, for both *TraWiC* and *PrismBench*, our studies can be expanded by using our frameworks for studying other non-transformer LLMs, such as diffusion language models.

- *LLMs are not trained only on code:* Given the rise of LLMs and their increasing prevalence in SE, we aimed to provide a more trustworthy pipeline by introducing *TraWiC* for dataset inclusion detection and *PrismBench* to provide a more comprehensive evaluation pipeline. However, unlike smaller code-specific models, LLMs are trained on extremely large datasets containing a mixture of natural language and code. As such, while *TraWiC* and *PrismBench* aim to support LLMs within the SE pipeline, they do not fully account for the complexity introduced by this mixed training dataset. Specifi-

cally, *TraWiC* focuses on semantic and syntactic identifiers extracted from code scripts for inclusion detection; however, these identifiers might have appeared in the models' training dataset within natural language contexts rather than code, potentially leading to false positives. Similarly, while *PrismBench* dynamically generates challenges to reduce memorization risks, the vast scale of LLM training datasets means that generated challenges may inadvertently resemble content the models encountered during pre-training, potentially resulting in inflated evaluation results for specific challenges and requiring multiple revisits, thereby increasing benchmarking costs.

- *Cost of analyses/evaluation:* Evaluating LLMs is expensive. Running large-scale experiments, whether through API or local deployment, can introduce substantial computational and financial costs. For both *TraWiC* and *PrismBench*, we attempt to reduce inference overhead by optimizing the number of necessary queries on LLMs. However, while we reduce costs by focusing on targeted, code-specific MIAs for *TraWiC* (see Chapter 5.2.4), and provide a lite-mode to reduce the number of LLM queries (see Chapter 6.4.4) for *PrismBench*, further optimizations are necessary considering the mixed training data limitations discussed above.

7.2 Future Research

The frameworks presented in this thesis form a set of complementary tools for building trustworthy ML systems for SE. However, as described above, training and deploying ML models is not a single, one-time process. Instead, it is continuous: models are regularly retrained, redeployed, and evaluated as new data becomes available, downstream tasks evolve, and system requirements shift. This necessitates continuous monitoring, deployment, and feedback. The contributions of this thesis represent a first step in that direction, and we hope that they can serve as a foundation for further extension and exploration by both researchers and practitioners. Based on our findings, we outline several promising avenues for future work:

- *Representation-aware training and probing at scale:* With *DeepCodeProbe* we demonstrated how well-chosen representations often matter more than sheer model scale for small and mid-size architectures. As we describe in detail in Chapters 3.1 and 4.2.1, many probing approaches have studied whether LLMs internalize programming languages' syntax. However, training LLMs is expensive. Future work can study whether incorporating probe-driven regularization, such as using syntax retention metrics as auxiliary loss, can improve training efficiency and convergence speed for code LLMs

and potentially lead to more cost-effective and targeted model development.

- *Sensitive information filtering: TraWiC* enables targeted end-to-end dataset inclusion detection regardless of the underlying LLM. An important aspect of LLM red-teaming is preventing models from reproducing sensitive information that may have been inadvertently included in training datasets, such as API keys, security credentials, or access protocols. Future research could extend *TraWiC*'s targeted detection approach to identify such sensitive content during pre-training checkpoints, thereby reducing the risk of incorporating confidential information into the model's learned parameters.

- *Evaluating LLM generalization:* Future research can integrate *TraWiC* and *PrismBench* to evaluate LLM's generalization capability. Specifically, *PrismBench* currently generates challenges dynamically and uses functional correctness as the evaluation objective. However, a hybrid approach could use a reference bank of pre-defined challenges alongside their solutions as the underlying benchmarking dataset. *TraWiC* could then be used to detect similarities between models' generated solutions/tests and this reference bank, enabling the search process to identify and prioritize scenarios where models exhibit memorization rather than genuine problem-solving capabilities.

- *Task evaluation guarantees:* An important aspect of *PrismBench*'s dynamic evaluation process is dynamic challenge generation to reduce the risk of inflated benchmarking results due to memorization. As detailed in Chapter 6.3.4, *PrismBench*'s MDP formalization, alongside repeated sampling per node, mitigates the risk of invalid or misaligned generated challenges compromising the benchmarking results. However, future research could optimize the required number of samples per node by integrating task evaluation frameworks such as DyVal [195] or TaskEval [313] to better validate challenge quality and, therefore, reduce the sampling overhead while maintaining evaluation integrity.

- *Self-refining training procedure: PrismBench* provides a detailed breakdown of *how* and *why* models fail at end-to-end programming challenges. Such failures across various stages of code generation (solution/test generation, program repair, instruction following) offer a natural mechanism for dataset augmentation. Future research could explore automated training loops in which *PrismBench* continuously evaluates a model, collects failure cases, and feeds these back into the training dataset. This closed-loop pipeline could support post-training or fine-tuning pipelines that iteratively focus on model failures and improve model performance until convergence.

- *Multi-objective evaluation: PrismBench* demonstrates how dynamic benchmarking, adaptive challenge generation, and isolating the evaluation pipelines can uncover sub-

tle and systematic failure modes in code generation. However, our current tree search focuses solely on functional correctness (i.e, the resulting code runs and passes tests). As such, future work could extend this to consider additional objectives, such as runtime performance, memory consumption, and security vulnerabilities, enabling more comprehensive assessments of LLMs' code generation capabilities.

REFERENCES

[1] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[2] A. Nair, A. Roy, and K. Meinke, "funcgnn: A graph neural network approach to program similarity," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–11.

[3] Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, and T. Matsumura, "Automatic source code summarization with extended tree-lstm," in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–8.

[4] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 397–407.

[5] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 511–523.

[6] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 341–353.

[7] N. D. Bui, Y. Yu, and L. Jiang, "Infercode: Self-supervised learning of code representations by predicting subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1186–1197.

[8] A. M. Mir, E. Latoškinas, S. Proksch, and G. Gousios, "Type4py: Practical deep similarity learning-based type inference for python," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2241–2252.

[9] G. Assres, G. Bhandari, A. Shalaginov, T.-M. Gronli, and G. Ghinea, "State-of-the-art and challenges of engineering ml-enabled software systems in the deep learning era," *ACM Computing Surveys*, vol. 57, no. 10, pp. 1–35, 2025.

[10] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–73, 2022.

[11] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[12] C. Rudin, "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead," *Nature machine intelligence*, vol. 1, no. 5, pp. 206–215, 2019.

[13] B. Xua and G. Yang, "Interpretability research of deep learning: A literature survey," *Information Fusion*, p. 102721, 2024.

[14] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *2017 3rd IEEE international conference on computer and communications (ICCC)*. IEEE, 2017, pp. 1298–1302.

[15] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "Vudenc: vulnerability detection with deep learning on a natural codebase for python," *Information and Software Technology*, vol. 144, p. 106809, 2022.

[16] B. Nyirongo, Y. Jiang, H. Jiang, and H. Liu, "A survey of deep learning based software refactoring," *arXiv preprint arXiv:2404.19226*, 2024.

[17] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*. PMLR, 2016, pp. 2091–2100.

[18] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th conference on program comprehension*, 2018, pp. 200–210.

[19] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 602–614.

[20] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[21] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[22] Anthropic, "Claude Opus 4 & Claude Sonnet 4 System Card," Technical report, May 2025. [Online]. Available: https://www-cdn.anthropic.com/6be99a52cb68eb70eb9572b4cafad13df32ed995.pdf

[23] OpenAI, "OpenAI O3 and O4-Mini System Card," Technical report, Apr. 16 2025. [Online]. Available: https://cdn.openai.com/pdf/2221c875-02dc-4789-800b-e7758f3722c1/o3-and-o4-mini-system-card.pdf

[24] M. Seo, J. Baek, S. Lee, and S. J. Hwang, "Paper2code: Automating code generation from scientific papers in machine learning," *arXiv preprint arXiv:2504.17192*, 2025.

[25] X. Liu, B. Lan, Z. Hu, Y. Liu, Z. Zhang, F. Wang, M. Shieh, and W. Zhou, "Codexgraph: Bridging large language models and code repositories via code graph databases," *arXiv preprint arXiv:2408.03910*, 2024.

[26] B. Sherifi, K. Slhoub, and F. Nembhard, "The potential of llms in automating software testing: From generation to reporting," *arXiv preprint arXiv:2501.00217*, 2024.

[27] M. A. Ferrag, N. Tihanyi, and M. Debbah, "From llm reasoning to autonomous ai agents: A comprehensive review," *arXiv preprint arXiv:2504.19678*, 2025.

[28] Amazon Web Services, "Amazon Q – Generative AI Assistant," https://aws.amazon.com/q/, 2025, accessed: 2025-06-29.

[29] OpenAI, "Introducing Codex," https://openai.com/index/introducing-codex/, 2025, accessed: 2025-06-29.

[30] H. Tao, Y. Chen, H. Wu, and R. Deng, "A survey of software trustworthiness measurements," *International Journal of Performability Engineering*, vol. 15, no. 9, p. 2364, 2019.

[31] R. S. Pressman, "Software engineering a practitioners approach," 2022.

[32] Y. Liu, F. Tian, Y. Meng, H. Beg, K. Jiang, R. Singh, W. Chen, and P. Sun. (2024) Meta's approach to machine learning prediction robustness. Engineering at Meta, published July 10, 2024. Accessed June 29, 2025. [Online]. Available: https://engineering.fb.com/2024/07/10/data-infrastructure/machine-learning-ml-prediction-robustness-meta/

[33] M. Tabachnyk, S. Nikolov *et al.*, "Ml-enhanced code completion improves developer productivity," *Google Research Blog.[Online]. Available: https://research. google/blog/ml-enhanced-code-completion-improves-developer-productivity*, 2022.

[34] L. Chen, Q. Guo, H. Jia, Z. Zeng, X. Wang, Y. Xu, J. Wu, Y. Wang, Q. Gao, J. Wang *et al.*, "A survey on evaluating large language models in code generation tasks," *arXiv preprint arXiv:2408.16498*, 2024.

[35] V. Majdinasab, A. Nikanjam, and F. Khomh, "Deepcodeprobe: Evaluating code representation quality in models trained on code," *Empirical Software Engineering*, vol. 30, no. 6, pp. 1–53, 2025.

[36] ——, "Trained without my consent: Detecting code inclusion in language models trained on code," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–46, 2025.

[37] ——, "Prism: Dynamic and flexible benchmarking of llms code generation with monte carlo tree search," *arXiv preprint arXiv:2504.05500*, 2025.

[38] T. M. Mitchell *et al.*, *Machine learning*. McGraw-hill New York, 2007, vol. 1.

[39] I. Muhammad and Z. Yan, "Supervised machine learning approaches: A survey." *ICTACT Journal on Soft Computing*, vol. 5, no. 3, 2015.

[40] M. E. Celebi and K. Aydin, *Unsupervised learning algorithms*. Springer, 2016, vol. 9.

[41] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[42] S. Dong, P. Wang, and K. Abbas, "A survey on deep learning and its applications," *Computer Science Review*, vol. 40, p. 100379, 2021.

[43] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: analysis, applications, and prospects," *IEEE transactions on neural networks and learning systems*, vol. 33, no. 12, pp. 6999–7019, 2021.

[44] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[45] V. Rajlich, "Software evolution and maintenance," in *Future of Software Engineering Proceedings*, 2014, pp. 133–144.

[46] S. Koch, “Evolution of open source software systems–a large-scale investigation,” in *Proceedings of the 1st International Conference on Open Source Systems*, 2005, pp. 148–153.

[47] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, “Challenges in software evolution,” in *Eighth International Workshop on Principles of Software Evolution (IWPSE’05)*. IEEE, 2005, pp. 13–22.

[48] H. C. Benestad, B. Anda, and E. Arisholm, “Understanding cost drivers of software evolution: a quantitative and qualitative investigation of change effort in two evolving software systems,” *Empirical Software Engineering*, vol. 15, pp. 166–203, 2010.

[49] R. Gupta, S. Pal, A. Kanade, and S. Shevade, “Deepfix: Fixing common c language errors by deep learning,” in *Proceedings of the aaai conference on artificial intelligence*, vol. 31, 2017.

[50] S. Bhatia, P. Kohli, and R. Singh, “Neuro-symbolic program corrector for introductory programming assignments,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 60–70.

[51] A. Barbez, F. Khomh, and Y.-G. Guéhéneuc, “Deep learning anti-patterns from code metrics history,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 114–124.

[52] D. Gros, H. Sezhiyan, P. Devanbu, and Z. Yu, “Code to comment" translation" data, metrics, baselining & evaluation,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 746–757.

[53] L. Kumar and S. K. Rath, “Hybrid functional link artificial neural network approach for predicting maintainability of object-oriented software,” *Journal of Systems and Software*, vol. 121, pp. 170–190, 2016.

[54] E. Dehaerne, B. Dey, S. Halder, S. De Gendt, and W. Meert, “Code generation using machine learning: A systematic review,” *Ieee Access*, vol. 10, pp. 82 434–82 455, 2022.

[55] H. Tong, B. Liu, and S. Wang, “Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning,” *Information and Software Technology*, vol. 96, pp. 94–111, 2018.

[56] H. K. Dam, T. Pham, S. W. Ng, T. Tran, J. Grundy, A. Ghose, T. Kim, and C.-J. Kim, “A deep tree-based model for software defect prediction,” *arXiv preprint arXiv:1802.00921*, 2018.

[57] Y. Liu, Y. Li, J. Guo, Y. Zhou, and B. Xu, "Connecting software metrics across versions to predict defects," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 232–243.

[58] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

[59] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 87–98.

[60] Y. Gao, Z. Wang, S. Liu, L. Yang, W. Sang, and Y. Cai, "Teccd: A tree embedding approach for code clone detection," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. ieee, 2019, pp. 145–156.

[61] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th international conference on program comprehension*, 2020, pp. 184–195.

[62] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[63] Z. Li, Y. Wu, B. Peng, X. Chen, Z. Sun, Y. Liu, and D. Paul, "Setransformer: A transformer-based code semantic parser for code comment generation," *IEEE Transactions on Reliability*, vol. 72, no. 1, pp. 258–273, 2022.

[64] P. D. Nagy, M. A. Najafabadi, and H. Davoudi, "A survey on code representation," *World Scientific Annual Review of Artificial Intelligence*, vol. 1, p. 2350001, 2023.

[65] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," *arXiv preprint arXiv:2007.04973*, 2020.

[66] Y. Peng, C. Gao, Z. Li, B. Gao, D. Lo, Q. Zhang, and M. Lyu, "Static inference meets deep learning: a hybrid type inference approach for python," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2019–2030.

[67] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, I. Vats, H. Moazen, and F. Sarro, "A survey on machine learning techniques for source code analysis," *arXiv preprint arXiv:2110.09610*, 2021.

[68] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[69] Z. Zhang and T. Saber, "Ast-enhanced or ast-overloaded? the surprising impact of hybrid graph representations on code clone detection," *arXiv preprint arXiv:2506.14470*, 2025.

[70] F. Ye, S. Zhou, A. Venkat, R. Marcus, P. Petersen, J. J. Tithi, T. Mattson, T. Kraska, P. Dubey, V. Sarkar *et al.*, "Context-aware parse trees," *arXiv preprint arXiv:2003.11118*, 2020.

[71] W. Sun, C. Fang, Y. Miao, Y. You, M. Yuan, Y. Chen, Q. Zhang, A. Guo, X. Chen, Y. Liu *et al.*, "Abstract syntax tree for programming language understanding and representation: How far are we?" *arXiv preprint arXiv:2312.00413*, 2023.

[72] R. Wu, Y. Zhang, Q. Peng, L. Chen, and Z. Zheng, "A survey of deep learning models for structural code understanding," *arXiv preprint arXiv:2205.01293*, 2022.

[73] K. C. Swarna, N. S. Mathews, D. Vagavolu, and S. Chimalakonda, "On the impact of multiple source code representations on software engineering tasks—an empirical study," *Journal of Systems and Software*, vol. 210, p. 111941, 2024.

[74] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[75] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[76] M. Lei, H. Li, J. Li, N. Aundhkar, and D.-K. Kim, "Deep learning application on code clone detection: A review of current knowledge," *Journal of Systems and Software*, vol. 184, p. 111141, 2022.

[77] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, "Fcca: Hybrid code representation for functional clone detection using attention networks," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 304–318, 2020.

[78] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 249–260.

[79] C. Zhang, J. Wang, Q. Zhou, T. Xu, K. Tang, H. Gui, and F. Liu, "A survey of automatic source code summarization," *Symmetry*, vol. 14, no. 3, p. 471, 2022.

[80] X. Song, H. Sun, X. Wang, and J. Yan, "A survey of automatic generation of source code comments: Algorithms and techniques," *IEEE Access*, vol. 7, pp. 111 411–111 428, 2019.

[81] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," *Advances in neural information processing systems*, vol. 32, 2019.

[82] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–69, 2023.

[83] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "Thinkrepair: Self-directed automated program repair," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1274–1286.

[84] T. Parr and K. Fisher, "Ll (*) the foundation of the antlr parser generator," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 425–436, 2011.

[85] D. Zhang, W. Ahmad, M. Tan, H. Ding, R. Nallapati, D. Roth, X. Ma, and B. Xiang, "Code representation learning at scale," *arXiv preprint arXiv:2402.01935*, 2024.

[86] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[87] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[88] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[89] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, "Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation," *arXiv preprint arXiv:2108.04556*, 2021.

[90] A. Jaiswal, A. R. Babu, M. Z. Zadeh, D. Banerjee, and F. Makedon, "A survey on contrastive self-supervised learning," *Technologies*, vol. 9, no. 1, p. 2, 2020.

[91] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[92] X. Li, Y. Gong, Y. Shen, X. Qiu, H. Zhang, B. Yao, W. Qi, D. Jiang, W. Chen, and N. Duan, "Coderetriever: A large scale contrastive pre-training method for code search," in *Proceedings of the 2022 conference on empirical methods in natural language processing*, 2022, pp. 2898–2910.

[93] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from" big code"," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 111–124, 2015.

[94] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 607–618.

[95] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 152–162.

[96] S. Choudhary, N. Chatterjee, and S. K. Saha, "Interpretation of black box nlp models: A survey," *arXiv preprint arXiv:2203.17081*, 2022.

[97] J. Hewitt and C. D. Manning, "A structural probe for finding syntax in word representations," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 4129–4138.

[98] Y. Yaghoobzadeh, K. Kann, T. J. Hazen, E. Agirre, and H. Schütze, "Probing for semantic classes: Diagnosing the meaning content of word embeddings," *arXiv preprint arXiv:1906.03608*, 2019.

[99] T. Pimentel, J. Valvoda, R. H. Maudslay, R. Zmigrod, A. Williams, and R. Cotterell, "Information-theoretic probing for linguistic structure," *arXiv preprint arXiv:2004.03061*, 2020.

[100] E. Goodwin, K. Sinha, and T. J. O'Donnell, "Probing linguistic systematicity," *arXiv preprint arXiv:2005.04315*, 2020.

[101] G. Vilone and L. Longo, "Explainable artificial intelligence: a systematic review," *arXiv preprint arXiv:2006.00093*, 2020.

[102] Y. Belinkov, "Probing classifiers: Promises, shortcomings, and advances," *Computational Linguistics*, vol. 48, no. 1, pp. 207–219, 2022.

[103] S. Luo, H. Ivison, S. C. Han, and J. Poon, "Local interpretations for explainable natural language processing: A survey," *ACM Computing Surveys*, 2021.

[104] R. Mitkov, R. Evans, C. Orăsan, I. Dornescu, and M. Rios, "Coreference resolution: To what extent does it help nlp applications?" in *Text, Speech and Dialogue: 15th International Conference, TSD 2012, Brno, Czech Republic, September 3-7, 2012. Proceedings 15*. Springer, 2012, pp. 16–27.

[105] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning (still) requires rethinking generalization," *Communications of the ACM*, vol. 64, no. 3, pp. 107–115, 2021.

[106] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE symposium on security and privacy (SP)*. IEEE, 2017, pp. 3–18.

[107] S. Yeom, I. Giacomelli, M. Fredrikson, and S. Jha, "Privacy risk in machine learning: Analyzing the connection to overfitting," in *2018 IEEE 31st computer security foundations symposium (CSF)*. IEEE, 2018, pp. 268–282.

[108] M. Zhu, C. Guo, C. Feng, and O. Simeone, "On the impact of uncertainty and calibration on likelihood-ratio membership inference attacks," *arXiv preprint arXiv:2402.10686*, 2024.

[109] L. Watson, C. Guo, G. Cormode, and A. Sablayrolles, "On the importance of difficulty calibration in membership inference attacks," *arXiv preprint arXiv:2111.08440*, 2021.

[110] Y. Long, V. Bindschaedler, L. Wang, D. Bu, X. Wang, H. Tang, C. A. Gunter, and K. Chen, "Understanding membership inferences on well-generalized learning models," *arXiv preprint arXiv:1802.04889*, 2018.

[111] S. Rezaei and X. Liu, "On the difficulty of membership inference attacks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 7892–7900.

[112] M. Nasr, N. Carlini, J. Hayase, M. Jagielski, A. F. Cooper, D. Ippolito, C. A. Choquette-Choo, E. Wallace, F. Tramèr, and K. Lee, "Scalable extraction of training data from (production) language models," *arXiv preprint arXiv:2311.17035*, 2023.

[113] N. Carlini, J. Hayes, M. Nasr, M. Jagielski, V. Sehwag, F. Tramer, B. Balle, D. Ippolito, and E. Wallace, "Extracting training data from diffusion models," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5253–5270.

[114] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang, "Pre-trained models for natural language processing: A survey," *Science China Technological Sciences*, vol. 63, no. 10, pp. 1872–1897, 2020.

[115] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," 2018.

[116] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[117] F. Petroni, T. Rocktäschel, P. Lewis, A. Bakhtin, Y. Wu, A. H. Miller, and S. Riedel, "Language models as knowledge bases?" *arXiv preprint arXiv:1909.01066*, 2019.

[118] J. Feldman, J. Davison, and A. M. Rush, "Commonsense knowledge mining from pretrained models," *arXiv preprint arXiv:1909.00505*, 2019.

[119] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[120] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[121] V. B. Parthasarathy, A. Zafar, A. Khan, and A. Shahid, "The ultimate guide to fine-tuning llms from basics to breakthroughs: An exhaustive review of technologies, research, best practices, applied research challenges and opportunities," *arXiv preprint arXiv:2408.13296*, 2024.

[122] Z. Shengyu, D. Linfeng, L. Xiaoya, Z. Sen, S. Xiaofei, W. Shuhe, L. Jiwei, R. Hu, Z. Tianwei, F. Wu *et al.*, "Instruction tuning for large language models: A survey," *arXiv preprint arXiv:2308.10792*, 2023.

[123] G. Tie, Z. Zhao, D. Song, F. Wei, R. Zhou, Y. Dai, W. Yin, Z. Yang, J. Yan, Y. Su *et al.*, "A survey on post-training of large language models," *arXiv e-prints*, pp. arXiv–2503, 2025.

[124] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[125] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey *et al.*, "Santacoder: don't reach for the stars!" *arXiv preprint arXiv:2301.03988*, 2023.

[126] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[127] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[128] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[129] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," *arXiv preprint arXiv:2306.08568*, 2023.

[130] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[131] V. Aryabumi, Y. Su, R. Ma, A. Morisot, I. Zhang, A. Locatelli, M. Fadaee, A. Üstün, and S. Hooker, "To code, or not to code? exploring impact of code in pre-training," *arXiv preprint arXiv:2408.10914*, 2024.

[132] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[133] Y. Jiang, Y. Zhang, L. Lu, C. Treude, X. Su, S. Huang, and T. Wang, "Enhancing high-quality code generation in large language models with comparative prefix-tuning," *arXiv preprint arXiv:2503.09020*, 2025.

[134] D. Kocetkov, R. Li, L. Ben Allal, J. Li, C. Mou, C. Muñoz Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, "The stack: 3 tb of permissively licensed source code," *Preprint*, 2022.

[135] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," *arXiv preprint arXiv:1606.05250*, 2016.

[136] T. Kwiatkowski, J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, J. Devlin, K. Lee *et al.*, "Natural questions: a benchmark for question answering research," *Transactions of the Association for Computational Linguistics*, vol. 7, pp. 453–466, 2019.

[137] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[138] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[139] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[140] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.

[141] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.

[142] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring massive multitask language understanding," *arXiv preprint arXiv:2009.03300*, 2020.

[143] S. Lin, J. Hilton, and O. Evans, "Truthfulqa: Measuring how models mimic human falsehoods," *arXiv preprint arXiv:2109.07958*, 2021.

[144] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

[145] P. Liang, R. Bommasani, T. Lee, D. Tsipras, D. Soylu, M. Yasunaga, Y. Zhang, D. Narayanan, Y. Wu, A. Kumar *et al.*, "Holistic evaluation of language models," *arXiv preprint arXiv:2211.09110*, 2022.

[146] A. Srivastava, A. Rastogi, A. Rao, A. A. M. Shoeb, A. Abid, A. Fisch, A. R. Brown, A. Santoro, A. Gupta, A. Garriga-Alonso *et al.*, "Beyond the imitation game: Quantifying and extrapolating the capabilities of language models," *arXiv preprint arXiv:2206.04615*, 2022.

[147] A. Jaech, A. Kalai, A. Lerer, A. Richardson, A. El-Kishky, A. Low, A. Helyar, A. Madry, A. Beutel, A. Carney *et al.*, "Openai o1 system card," *arXiv preprint arXiv:2412.16720*, 2024.

[148] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[149] C. Xu, S. Guan, D. Greene, M. Kechadi *et al.*, "Benchmark data contamination of large language models: A survey," *arXiv preprint arXiv:2406.04244*, 2024.

[150] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, H. Zhang, B. Zhu, M. Jordan, J. E. Gonzalez *et al.*, "Chatbot arena: An open platform for evaluating llms by human preference," *arXiv preprint arXiv:2403.04132*, 2024.

[151] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46 595–46 623, 2023.

[152] Y. Lu, M. Bartolo, A. Moore, S. Riedel, and P. Stenetorp, "Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity," *arXiv preprint arXiv:2104.08786*, 2021.

[153] A. Rogers, O. Kovaleva, and A. Rumshisky, "A primer in bertology: What we know about how bert works," *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 842–866, 2021.

[154] N. F. Liu, M. Gardner, Y. Belinkov, M. E. Peters, and N. A. Smith, "Linguistic knowledge and transferability of contextual representations," *arXiv preprint arXiv:1903.08855*, 2019.

[155] A. Miaschi and F. Dell'Orletta, "Contextual and non-contextual word embeddings: an in-depth linguistic investigation," in *Proceedings of the 5th Workshop on Representation Learning for NLP*, 2020, pp. 110–119.

[156] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture? a structural analysis of pre-trained language models for source code," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2377–2388.

[157] S. Troshin and N. Chirkova, "Probing pretrained models of source code," *arXiv preprint arXiv:2202.08975*, 2022.

[158] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?" in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1332–1336.

[159] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[160] W. Ma, S. Liu, M. Zhao, X. Xie, W. Wang, Q. Hu, J. Zhang, and Y. Liu, "Unveiling code pre-trained models: Investigating syntax and semantics capacities," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–29, 2024.

[161] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[162] A. Karmakar and R. Robbes, "Inspect: Intrinsic and systematic probing evaluation for code transformers," *IEEE Transactions on Software Engineering*, vol. 50, no. 2, pp. 220–238, 2023.

[163] J. Jia, A. Salem, M. Backes, Y. Zhang, and N. Z. Gong, "Memguard: Defending against black-box membership inference attacks via adversarial examples," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 259–274.

[164] H. Hu, Z. Salcic, G. Dobbie, Y. Chen, and X. Zhang, "Ear: An enhanced adversarial regularization approach against membership inference attacks," in *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2021, pp. 1–8.

[165] B. Hui, Y. Yang, H. Yuan, P. Burlina, N. Z. Gong, and Y. Cao, "Practical blind membership inference attack via differential comparisons," *arXiv preprint arXiv:2101.01341*, 2021.

[166] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson *et al.*, "Extracting training data from large language models," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2633–2650.

[167] N. Carlini, D. Ippolito, M. Jagielski, K. Lee, F. Tramer, and C. Zhang, "Quantifying memorization across neural language models," *arXiv preprint arXiv:2202.07646*, 2022.

[168] K. K. Chang, M. Cramer, S. Soni, and D. Bamman, "Speak, memory: An archaeology of books known to chatgpt/gpt-4," *arXiv preprint arXiv:2305.00118*, 2023.

[169] C. Zhang, D. Ippolito, K. Lee, M. Jagielski, F. Tramèr, and N. Carlini, "Counterfactual memorization in neural language models," *arXiv preprint arXiv:2112.12938*, 2021.

[170] D. Choi, Y. Shavit, and D. Duvenaud, "Tools for verifying neural models' training data," *arXiv preprint arXiv:2307.00682*, 2023.

[171] S. Zhang and H. Li, "Code membership inference for detecting unauthorized data use in code pre-trained language models," *arXiv preprint arXiv:2312.07200*, 2023.

[172] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, "Gotcha! this model uses my code! evaluating membership leakage risks in code models," *arXiv preprint arXiv:2310.01166*, 2023.

[173] R. Xu, Z. Wang, R.-Z. Fan, and P. Liu, "Benchmarking benchmark leakage in large language models," *arXiv preprint arXiv:2404.18824*, 2024.

[174] M. Roberts, H. Thakur, C. Herlihy, C. White, and S. Dooley, "Data contamination through the lens of time," *arXiv preprint arXiv:2310.10628*, 2023.

[175] M. Jiang, K. Z. Liu, M. Zhong, R. Schaeffer, S. Ouyang, J. Han, and S. Koyejo, "Investigating data contamination for pre-training language models," *arXiv preprint arXiv:2401.06059*, 2024.

[176] S. Balloccu, P. Schmidtová, M. Lango, and O. Dušek, "Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source llms," *arXiv preprint arXiv:2402.03927*, 2024.

[177] Y. Dong, X. Jiang, H. Liu, Z. Jin, B. Gu, M. Yang, and G. Li, "Generalization or memorization: Data contamination and trustworthy evaluation for large language models," *arXiv preprint arXiv:2402.15938*, 2024.

[178] C. Fourrier, N. Habib, A. Lozovskaya, K. Szafer, and T. Wolf, "Open llm leaderboard v2," https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard, 2024.

[179] B. Y. Lin, Y. Deng, K. Chandu, F. Brahman, A. Ravichander, V. Pyatkin, N. Dziri, R. L. Bras, and Y. Choi, "Wildbench: Benchmarking llms with challenging tasks from real users in the wild," *arXiv preprint arXiv:2406.04770*, 2024.

[180] F. Chollet, "On the measure of intelligence," *arXiv preprint arXiv:1911.01547*, 2019.

[181] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=VTF8yNQM66

[182] P. Liang, R. Bommasani, T. Lee, D. Tsipras, D. Soylu, M. Yasunaga, Y. Zhang, D. Narayanan, Y. Wu, A. Kumar, B. Newman, B. Yuan, B. Yan, C. Zhang, C. A. Cosgrove, C. D. Manning, C. Re, D. Acosta-Navas, D. A. Hudson, E. Zelikman, E. Durmus, F. Ladhak, F. Rong, H. Ren, H. Yao, J. WANG, K. Santhanam, L. Orr, L. Zheng, M. Yuksekgonul, M. Suzgun, N. Kim, N. Guha, N. S. Chatterji, O. Khattab, P. Henderson, Q. Huang, R. A. Chi, S. M. Xie, S. Santurkar, S. Ganguli, T. Hashimoto, T. Icard, T. Zhang, V. Chaudhary, W. Wang, X. Li, Y. Mai, Y. Zhang, and Y. Koreeda, "Holistic evaluation of language models," *Transactions on Machine Learning Research*, 2023, featured Certification, Expert Certification. [Online]. Available: https://openreview.net/forum?id=iO4LZibEqW

[183] L. Phan, A. Gatti, Z. Han, N. Li, J. Hu, H. Zhang, S. Shi, M. Choi, A. Agrawal, A. Chopra *et al.*, "Humanity's last exam," *arXiv*, 2025.

[184] C. White, S. Dooley, M. Roberts, A. Pal, B. Feuer, S. Jain, R. Shwartz-Ziv, N. Jain, K. Saifullah, S. Naidu *et al.*, "Livebench: A challenging, contamination-free llm benchmark," *arXiv preprint arXiv:2406.19314*, 2024.

[185] T. Franzmeyer, A. Shtedritski, S. Albanie, P. Torr, J. F. Henriques, and J. N. Foerster, "Hellofresh: Llm evaluations on streams of real-world human editorial actions across x community notes and wikipedia edits," *arXiv preprint arXiv:2406.03428*, 2024.

[186] Anthropic, "The claude 3 model family: Opus, sonnet, haiku," https://docs.anthropic.com/en/docs/resources/model-card, 2024, accessed: 2025-01-12.

[187] L. Zhu, X. Wang, and X. Wang, "Judgelm: Fine-tuned large language models are scalable judges," *arXiv preprint arXiv:2310.17631*, 2023.

[188] X. Li, T. Zhang, Y. Dubois, R. Taori, I. Gulrajani, C. Guestrin, P. Liang, and T. B. Hashimoto, "Alpacaeval: An automatic evaluator of instruction-following models," https://github.com/tatsu-lab/alpaca_eval, 5 2023.

[189] T. Wang, P. Yu, X. E. Tan, S. O'Brien, R. Pasunuru, J. Dwivedi-Yu, O. Golovneva, L. Zettlemoyer, M. Fazel-Zarandi, and A. Celikyilmaz, "Shepherd: A critic for language model generation," *arXiv preprint arXiv:2308.04592*, 2023.

[190] A. S. Thakur, K. Choudhary, V. S. Ramayapally, S. Vaidyanathan, and D. Hupkes, "Judging the judges: Evaluating alignment and vulnerabilities in llms-as-judges," *arXiv preprint arXiv:2406.12624*, 2024.

[191] L. Li, B. Dong, R. Wang, X. Hu, W. Zuo, D. Lin, Y. Qiao, and J. Shao, "Salad-bench: A hierarchical and comprehensive safety benchmark for large language models," *arXiv preprint arXiv:2402.05044*, 2024.

[192] B. Zhang, K. Zhou, X. Wei, X. Zhao, J. Sha, S. Wang, and J.-R. Wen, "Evaluating and improving tool-augmented computation-intensive math reasoning," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[193] M. Zhuge, C. Zhao, D. Ashley, W. Wang, D. Khizbullin, Y. Xiong, Z. Liu, E. Chang, R. Krishnamoorthi, Y. Tian *et al.*, "Agent-as-a-judge: Evaluate agents with agents," *arXiv preprint arXiv:2410.10934*, 2024.

[194] X. Li, Y. Lan, and C. Yang, "Treeeval: Benchmark-free evaluation of large language models through tree planning," *arXiv preprint arXiv:2402.13125*, 2024.

[195] K. Zhu, J. Wang, Q. Zhao, R. Xu, and X. Xie, "Dyval 2: Dynamic evaluation of large language models by meta probing agents," *arXiv preprint arXiv:2402.14865*, 2024.

[196] ——, "Dynamic evaluation of large language models by meta probing agents," in *Forty-first International Conference on Machine Learning*, 2024.

[197] L. Fan, W. Hua, L. Li, H. Ling, and Y. Zhang, "Nphardeval: Dynamic benchmark on reasoning ability of large language models via complexity classes," *arXiv preprint arXiv:2312.14890*, 2023.

[198] S. Wang, Z. Long, Z. Fan, Z. Wei, and X. Huang, "Benchmark self-evolving: A multi-agent framework for dynamic llm evaluation," *arXiv preprint arXiv:2402.11443*, 2024.

[199] Z. Zhang, J. Chen, and D. Yang, "Darg: Dynamic evaluation of large language models via adaptive reasoning graph," *arXiv preprint arXiv:2406.17271*, 2024.

[200] R. E. Blackwell, J. Barry, and A. G. Cohn, "Towards reproducible llm evaluation: Quantifying uncertainty in llm benchmark scores," *arXiv preprint arXiv:2410.03492*, 2024.

[201] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," *Advances in neural information processing systems*, vol. 28, 2015.

[202] L. Budach, M. Feuerpfeil, N. Ihde, A. Nathansen, N. Noack, H. Patzlaff, F. Naumann, and H. Harmouch, "The effects of data quality on machine learning performance," *arXiv preprint arXiv:2207.14529*, 2022.

[203] A. Sanyal, V. Chatterji, N. Vyas, B. Epstein, N. Demir, and A. Corletti, "Fix your models by fixing your datasets," *arXiv preprint arXiv:2112.07844*, 2021.

[204] D. V. Carvalho, E. M. Pereira, and J. S. Cardoso, "Machine learning interpretability: A survey on methods and metrics," *Electronics*, vol. 8, no. 8, p. 832, 2019.

[205] N. Tang, M. Chen, Z. Ning, A. Bansal, Y. Huang, C. McMillan, and T. J.-J. Li, "A study on developer behaviors for validating and repairing llm-generated code using eye tracking and ide actions," *arXiv preprint arXiv:2405.16081*, 2024.

[206] X. Zhou, P. Liang, B. Zhang, Z. Li, A. Ahmad, M. Shahin, and M. Waseem, "On the concerns of developers when using github copilot," *arXiv preprint arXiv:2311.01020*, 2023.

[207] Y. Zhou, F. Tu, K. Sha, J. Ding, and H. Chen, "A survey on data quality dimensions and tools for machine learning," *arXiv preprint arXiv:2406.19614*, 2024.

[208] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[209] F. Tambon, A. M. Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and G. Antoniol, "Bugs in large language models generated code," *arXiv preprint arXiv:2403.08937*, 2024.

[210] Y. Chen, G. Ou, M. Liu, Y. Wang, and Z. Zheng, "Are decoder-only large language models the silver bullet for code search?" *arXiv preprint arXiv:2410.22240*, 2024.

[211] M. B. Moumoula, A. K. Kabore, J. Klein, and T. Bissyande, "Large language models for cross-language code clone detection," *arXiv preprint arXiv:2408.04430*, 2024.

[212] J. A. Hernández López, M. Weyssow, J. S. Cuadrado, and H. Sahraoui, "Ast-probe: Recovering abstract syntax trees from hidden representations of pre-trained language models," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–11.

[213] I. M. Johnstone and D. M. Titterington, "Statistical challenges of high-dimensional data," pp. 4237–4253, 2009.

[214] R. Vandaele, B. Kang, T. De Bie, and Y. Saeys, "The curse revisited: When are distances informative for the ground truth in noisy high-dimensional data?" in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2022, pp. 2158–2172.

[215] M. Chilowicz, E. Duris, and G. Roussel, "Syntax tree fingerprinting for source code similarity detection," in *2009 IEEE 17th international conference on program comprehension*. IEEE, 2009, pp. 243–247.

[216] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 542–553.

[217] J. Svacina, J. Simmons, and T. Cerny, "Semantic code clone detection for enterprise applications," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 129–131.

[218] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," *IEEE access*, vol. 7, pp. 86 121–86 144, 2019.

[219] G. d'Eon, J. d'Eon, J. R. Wright, and K. Leyton-Brown, "The spotlight: A general method for discovering systematic errors in deep learning models," in *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency*, 2022, pp. 1962–1981.

[220] N. Sohoni, J. Dunnmon, G. Angus, A. Gu, and C. Ré, "No subclass left behind: Fine-grained robustness in coarse-grained classification problems," *Advances in Neural Information Processing Systems*, vol. 33, pp. 19 339–19 352, 2020.

[221] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[222] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 476–480.

[223] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[224] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[225] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[226] N. D. Bui, Y. Yu, and L. Jiang, "Treecaps: Tree-based capsule networks for source code processing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 1, 2021, pp. 30–38.

[227] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 819–830.

[228] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural cnn decoder for code generation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 7055–7062.

[229] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.

[230] V. Majdinasab, "Deepcodeprobe," https://github.com/CommissarSilver/DeepCodeProbe, 2024.

[231] R. H. Maudslay, J. Valvoda, T. Pimentel, A. Williams, and R. Cotterell, "A tale of a probe and a parser," *arXiv preprint arXiv:2005.01641*, 2020.

[232] J. Hewitt and P. Liang, "Designing and interpreting probes with control tasks," *arXiv preprint arXiv:1909.03368*, 2019.

[233] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[234] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," *arXiv preprint arXiv:2001.08361*, 2020.

[235] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark *et al.*, "Training compute-optimal large language models," *arXiv preprint arXiv:2203.15556*, 2022.

[236] J. Lin, H. Dong, Y. Xie, and L. Zhang, "Scaling laws behind code understanding model," *arXiv preprint arXiv:2402.12813*, 2024.

[237] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou, "Deep learning scaling is predictable, empirically," *arXiv preprint arXiv:1712.00409*, 2017.

[238] J. S. Rosenfeld, A. Rosenfeld, Y. Belinkov, and N. Shavit, "A constructive prediction of the generalization error across scales," *arXiv preprint arXiv:1909.12673*, 2019.

[239] D. Hernandez and T. B. Brown, "Measuring the algorithmic efficiency of neural networks," *arXiv preprint arXiv:2005.04305*, 2020.

[240] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[241] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," *arXiv preprint arXiv:1611.03530*, 2016.

[242] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, "An empirical study of common challenges in developing deep learning applications," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 104–115.

[243] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, "A comprehensive study on challenges in deploying deep learning based software," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 750–762.

[244] H. Zhao, H. Chen, F. Yang, N. Liu, H. Deng, H. Cai, S. Wang, D. Yin, and M. Du, "Explainability for large language models: A survey," *ACM Transactions on Intelligent Systems and Technology*, vol. 15, no. 2, pp. 1–38, 2024.

[245] O. Sharir, B. Peleg, and Y. Shoham, "The cost of training nlp models: A concise overview," *arXiv preprint arXiv:2004.08900*, 2020.

[246] Y. Zhang, Y. Li, L. Cui, D. Cai, L. Liu, T. Fu, X. Huang, E. Zhao, Y. Zhang, Y. Chen *et al.*, "Siren's song in the ai ocean: a survey on hallucination in large language models," *arXiv preprint arXiv:2309.01219*, 2023.

[247] C. Ralhan and N. Malik, "A study of software clone detection techniques for better software maintenance and reliability," in *2021 International Conference on Computing Sciences (ICCS)*. IEEE, 2021, pp. 249–253.

[248] N. Saini, S. Singh *et al.*, "Code clones: Detection and management," *Procedia computer science*, vol. 132, pp. 718–727, 2018.

[249] A. W. Services. (2023) Code references - codewhisperer. [Online]. Available: https://docs.aws.amazon.com/codewhisperer/latest/userguide/code-reference.html

[250] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *2008 16th iEEE international conference on program comprehension*. IEEE, 2008, pp. 172–181.

[251] M. J. Wise, "String similarity via greedy string tiling and running karp-rabin matching," *Online Preprint, Dec*, vol. 119, no. 1, pp. 1–17, 1993.

[252] K. Tirumala, A. Markosyan, L. Zettlemoyer, and A. Aghajanyan, "Memorization without overfitting: Analyzing the training dynamics of large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 38 274–38 290, 2022.

[253] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[254] A. T. Ying, J. L. Wright, and S. Abrams, "Source code that talks: an exploration of eclipse task comments and their implication to repository mining," *ACM SIGSOFT software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[255] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, 2007, pp. 70–79.

[256] D. G. Feitelson, A. Mizrahi, N. Noy, A. B. Shabat, O. Eliyahu, and R. Sheffer, "How developers choose names," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 37–52, 2020.

[257] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1199–1210.

[258] (2023) Pep 8 – style guide for python code | peps.python.org. [Online]. Available: https://peps.python.org/pep-0008/

[259] (2023) Common lisp style guide | common lisp. [Online]. Available: https://lisp-lang.org/style-guide/

[260] E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd, "Software documentation: the practitioners' perspective," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 590–601.

[261] D. Haouari, H. Sahraoui, and P. Langlais, "How good is your comment? a study of comments in java programs," in *2011 International symposium on empirical software engineering and measurement*. IEEE, 2011, pp. 137–146.

[262] M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. McLeavey, J. Tworek, and M. Chen, "Efficient training of language models to fill in the middle," 2022.

[263] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, Feb. 1966.

[264] J. Wang and Y. Dong, "Measurement of text similarity: a survey," *Information*, vol. 11, no. 9, p. 421, 2020.

[265] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[266] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," *arXiv preprint arXiv:2305.13245*, 2023.

[267] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[268] S. Minaee, T. Mikolov, N. Nikzad, M. Chenaghlu, R. Socher, X. Amatriain, and J. Gao, "Large language models: A survey," *arXiv preprint arXiv:2402.06196*, 2024.

[269] M. A. A. Mamun, C. Berger, and J. Hansson, "Correlations of software code metrics: an empirical study," in *Proceedings of the 27th international workshop on software measurement and 12th international conference on software process and product measurement*, 2017, pp. 255–266.

[270] V. Majdinasab, "Trawic," https://github.com/CommissarSilver/TraWiC, 2024.

[271] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[272] N. Ding, Y. Qin, G. Yang, F. Wei, Z. Yang, Y. Su, S. Hu, Y. Chen, C.-M. Chan, W. Chen *et al.*, "Parameter-efficient fine-tuning of large-scale pre-trained language models," *Nature Machine Intelligence*, vol. 5, no. 3, pp. 220–235, 2023.

[273] L. Prechelt, G. Malpohl, M. Philippsen *et al.*, "Finding plagiarisms among a set of programs with jplag." *J. Univers. Comput. Sci.*, vol. 8, no. 11, p. 1016, 2002.

[274] M. Ciniselli, L. Pascarella, and G. Bavota, "To what extent do deep learning-based code recommenders generate predictions by cloning code from the training set?" in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 167–178.

[275] H. Hu, Z. Salcic, L. Sun, G. Dobbie, P. S. Yu, and X. Zhang, "Membership inference attacks on machine learning: A survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–37, 2022.

[276] V. Shejwalkar and A. Houmansadr, "Membership privacy for machine learning models through knowledge transfer," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, no. 11, 2021, pp. 9549–9557.

[277] J. Zheng, Y. Cao, and H. Wang, "Resisting membership inference attacks through knowledge distillation," *Neurocomputing*, vol. 452, pp. 114–126, 2021.

[278] C. Xu, J. Ren, D. Zhang, Y. Zhang, Z. Qin, and K. Ren, "Ganobfuscator: Mitigating information leakage under gan via differential privacy," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 9, pp. 2358–2371, 2019.

[279] L. Rosenblatt, X. Liu, S. Pouyanfar, E. de Leon, A. Desai, and J. Allen, "Differentially private synthetic data: Applied evaluations and enhancements," *arXiv preprint arXiv:2011.05537*, 2020.

[280] R. David, L. Coniglio, and M. Ceccato, "Qsynth-a program synthesis based approach for binary code deobfuscation," in *BAR 2020 Workshop*, 2020.

[281] D. Xu, J. Ming, and D. Wu, "Generalized dynamic opaque predicates: A new control flow obfuscation method," in *Information Security: 19th International Conference, ISC 2016, Honolulu, HI, USA, September 3-6, 2016. Proceedings 19*. Springer, 2016, pp. 323–342.

[282] S. A. Sebastian, S. Malgaonkar, P. Shah, M. Kapoor, and T. Parekhji, "A study & review on code obfuscation," in *2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave)*. IEEE, 2016, pp. 1–6.

[283] S. Kang, S. Lee, Y. Kim, S.-K. Mok, and E.-S. Cho, "Obfus: An obfuscation tool for software copyright and vulnerability protection," in *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, 2021, pp. 309–311.

[284] A. Ghosh, N. Manwani, and P. Sastry, "On the robustness of decision tree learning under label noise," in *Advances in Knowledge Discovery and Data Mining: 21st Pacific-Asia Conference, PAKDD 2017, Jeju, South Korea, May 23-26, 2017, Proceedings, Part I 21*. Springer, 2017, pp. 685–697.

[285] Y. Zhuang, "The performance cost of software obfuscation for android applications," *Computers & Security*, vol. 73, pp. 57–72, 2018.

[286] C. Bunse, "On the impact of code obfuscation to software energy consumption," in *From Science to Society: New Trends in Environmental Informatics*. Springer, 2018, pp. 239–249.

[287] P. Zhang, C. Wu, M. Peng, K. Zeng, D. Yu, Y. Lai, Y. Kang, W. Wang, and Z. Wang, "Khaos: The impact of inter-procedural code obfuscation on binary diffing techniques," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, 2023, pp. 55–67.

[288] J. H. Zar, "Spearman rank correlation," *Encyclopedia of biostatistics*, vol. 7, 2005.

[289] L. Breiman, *Classification and regression trees*. Routledge, 2017.

[290] T. R. McIntosh, T. Susnjak, N. Arachchilage, T. Liu, P. Watters, and M. N. Halgamuge, "Inadequacies of large language model benchmarks in the era of generative artificial intelligence," *arXiv preprint arXiv:2402.09880*, 2024.

[291] S. Banerjee, A. Agarwal, and E. Singh, "The vulnerability of language model benchmarks: Do they accurately reflect true llm performance?" *arXiv preprint arXiv:2412.03597*, 2024.

[292] F. Tambon, A. Nikanjam, F. Khomh, and G. Antoniol, "Assessing programming task difficulty for efficient evaluation of large language models," *arXiv preprint arXiv:2407.21227*, 2024.

[293] K. Zhou, Y. Zhu, Z. Chen, W. Chen, W. X. Zhao, X. Chen, Y. Lin, J.-R. Wen, and J. Han, "Don't make your llm an evaluation benchmark cheater," *arXiv preprint arXiv:2311.01964*, 2023.

[294] H. Li, Q. Dong, J. Chen, H. Su, Y. Zhou, Q. Ai, Z. Ye, and Y. Liu, "Llms-as-judges: A comprehensive survey on llm-based evaluation methods," *arXiv preprint arXiv:2412.05579*, 2024.

[295] D. Kiela, M. Bartolo, Y. Nie, D. Kaushik, A. Geiger, Z. Wu, B. Vidgen, G. Prasad, A. Singh, P. Ringshia *et al.*, "Dynabench: Rethinking benchmarking in nlp," *arXiv preprint arXiv:2104.14337*, 2021.

[296] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Pearson, 2016.

[297] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[298] Y. Shavit, S. Agarwal, M. Brundage, S. Adler, C. O'Keefe, R. Campbell, T. Lee, P. Mishkin, T. Eloundou, A. Hickey *et al.*, "Practices for governing agentic ai systems," *Research Paper, OpenAI, December*, 2023.

[299] Anthropic, "Building effective agents," December 19 2024, accessed: 2025-01-13. [Online]. Available: https://www.anthropic.com/research/building-effective-agents

[300] A. Roucher, T. Wolf, L. von Werra, and E. Kaunismäki, "'smolagents': The easiest way to build efficient agentic systems." https://github.com/huggingface/smolagents, 2025.

[301] V. Dibia, A. Fourney, G. Bansal, F. Poursabzi-Sangdeh, H. Liu, and S. Amershi, "Aligning offline metrics and human judgments of value for code generation models," *arXiv preprint arXiv:2210.16494*, 2022.

[302] Y. Zhang, Y. Xie, S. Li, K. Liu, C. Wang, Z. Jia, X. Huang, J. Song, C. Luo, Z. Zheng *et al.*, "Unseen horizons: Unveiling the real capability of llm code generation beyond the familiar," *arXiv preprint arXiv:2412.08109*, 2024.

[303] K. Li and Y. Yuan, "Large language models as test case generators: Performance evaluation and enhancement," *arXiv preprint arXiv:2404.13340*, 2024.

[304] C. Koutcheme, N. Dainese, and A. Hellas, "Using program repair as a proxy for language models' feedback ability in programming education," in *Workshop on Innovative Use of NLP for Building Educational Applications*. Association for Computational Linguistics, 2024, pp. 165–181.

[305] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, "Prompt engineering in large language models," in *International conference on data intelligence and cognitive informatics*. Springer, 2023, pp. 387–402.

[306] OpenAI, "Prompt engineering guide," https://platform.openai.com/docs/guides/prompt-engineering, 2025, accessed: 2025-01-22.

[307] Anthropic, "Prompt engineering overview," https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/overview, 2025, accessed: 2025-01-12.

[308] Anonymous, "Replication package for prismbench," https://github.com/PrismBench/PrismBench/tree/replication-package, 2025, accessed: 2025-05-15.

[309] M. Renze and E. Guven, "The effect of sampling temperature on problem solving in large language models," *arXiv preprint arXiv:2402.05201*, 2024.

[310] M. Peeperkorn, T. Kouwenhoven, D. Brown, and A. Jordanous, "Is temperature the creativity parameter of large language models?" *arXiv preprint arXiv:2405.00492*, 2024.

[311] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[312] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "Llm is like a box of chocolates: the non-determinism of chatgpt in code generation," *arXiv preprint arXiv:2308.02828*, 2023.

[313] F. Tambon, A. Nikanjam, C. Zid, F. Khomh, and G. Antoniol, "Taskeval: Assessing difficulty of code generation tasks for large language models," *arXiv preprint arXiv:2407.21227*, 2024.

[314] K. Zhu, J. Chen, J. Wang, N. Z. Gong, D. Yang, and X. Xie, "Dyval: Dynamic evaluation of large language models for reasoning tasks," *arXiv preprint arXiv:2309.17167*, 2023.

[315] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford *et al.*, "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[316] S. Agarwal, L. Ahmad, J. Ai, S. Altman, A. Applebaum, E. Arbus, R. K. Arora, Y. Bai, B. Baker, H. Bao *et al.*, "gpt-oss-120b & gpt-oss-20b model card," *arXiv preprint arXiv:2508.10925*, 2025.

[317] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[318] A. Meta, "The llama 4 herd: The beginning of a new era of natively multimodal ai innovation," *https://ai. meta. com/blog/llama-4-multimodal-intelligence/, checked on*, vol. 4, no. 7, p. 2025, 2025.

[319] LeetCode, "Programming challenges," December 19 2024, accessed: 2025-01-13. [Online]. Available: https://www.leetcode.com

[320] S. C. Y. Ho, V. Majdinasab, M. Islam, D. E. Costa, E. Shihab, F. Khomh, S. Nadi, and M. Raza, "An empirical study on bugs inside pytorch: A replication study," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023, pp. 220–231.

[321] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "Github copilot ai pair programmer: Asset or liability?" *Journal of Systems and Software*, vol. 203, p. 111734, 2023.

[322] F. Tambon, V. Majdinasab, A. Nikanjam, F. Khomh, and G. Antoniol, "Mutation testing of deep reinforcement learning based on real faults," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2023, pp. 188–198.

[323] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," *Information and Software Technology*, vol. 171, p. 107468, 2024.

[324] V. Majdinasab, M. J. Bishop, S. Rasheed, A. Moradidakhel, A. Tahir, and F. Khomh, "Assessing the security of github copilot's generated code-a targeted replication study," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024, pp. 435–444.

CO-AUTHORSHIP

The following research papers have been produced during my PhD. The following works were used in this thesis:

- **Majdinasab, V.**, Nikanjam, A., & Khomh, F. (2025). DeepCodeProbe: Evaluating Code Representation Quality in Models Trained on Code. Empirical Software Engineering, 30(6), 1-53. [35]
- **Majdinasab, V.**, Nikanjam, A., & Khomh, F. (2025). Trained without my consent: Detecting code inclusion in language models trained on code. ACM Transactions on Software Engineering and Methodology, 34(4), 1-46. [36]
- **Majdinasab, V.**, Nikanjam, A., & Khomh, F. (2025). PrismBench: Dynamic and Flexible Benchmarking of LLMs Code Generation with Monte Carlo Tree Search. Submitted to Advances in Neural Information Processing Systems (TMLR). [37]

While the following publications are also focused on ML for SE, they did not fit within the narrative of this thesis and were excluded:

- Ho, S. C. Y., **Majdinasab, V.**, Islam, M., Costa, D. E., Shihab, E., Khomh, F., ... & Raza, M. (2023, October). An empirical study on bugs inside pytorch: A replication study. In 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 220-231). IEEE. [320]
- Dakhel, A. M., **Majdinasab, V.**, Nikanjam, A., Khomh, F., Desmarais, M. C., & Jiang, Z. M. J. (2023). Github copilot ai pair programmer: Asset or liability?. Journal of Systems and Software, 203, 111734. [321]
- Tambon, F., **Majdinasab, V.**, Nikanjam, A., Khomh, F., & Antoniol, G. (2023, April). Mutation testing of deep reinforcement learning based on real faults. In 2023 IEEE Conference on Software Testing, Verification and Validation (ICST) (pp. 188-198). IEEE. [322]
- Dakhel, A. M., Nikanjam, A., **Majdinasab, V.**, Khomh, F., & Desmarais, M. C. (2024). Effective test generation using pre-trained large language models and mutation testing. Information and Software Technology, 171, 107468. [323]

- **Majdinasab, V.**, Bishop, M. J., Rasheed, S., Moradidakhel, A., Tahir, A., & Khomh, F. (2024, March). Assessing the Security of GitHub Copilot's Generated Code-A Targeted Replication Study. In 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 435-444). IEEE. [324]