Titre: Title:	Towards Reliable and Trustworthy Pipelines for MLOps and LLMOps
Auteur: Author:	Altaf Allah Abbassi
Date:	2025
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Abbassi, A. A. (2025). Towards Reliable and Trustworthy Pipelines for MLOps and LLMOps [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/64774/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/64774/
Directeurs de recherche: Advisors:	Foutse Khomh
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Towards Reliable and Trustworthy Pipelines for MLOps and LLMOps

ALTAF ALLAH ABBASSI
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Avril 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Towards Reliable and Trustworthy Pipelines for MLOps and LLMOps

présenté par **Altaf Allah ABBASSI**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Michel DESMARAIS, président
Foutse KHOMH, membre et directeur de recherche
Maxime LAMOTHE, membre

DEDICATION

To my family

. . .

ACKNOWLEDGEMENTS

First, I would like to express my profound gratitude to Pr. Foutse Khomh, who provided me with the opportunity to enter the world of research, for his guidance and patience throughout the Master's program, as well as for all of his feedback and support, which led to the realization of this thesis. I am especially thankful for the trust he placed in me, which empowered and motivated me to pursue my ideas.

Second, I want to thank Dr. Houssem Ben Braiek, whose feedback, knowledge, and research experience have been invaluable to me. He guided me in learning how to conduct research and supported me in writing my first paper.

Thirdly, I'd like to express my gratitude to Dr. Leuson Da Silva for his unwavering support and patience throughout this journey. He guided me and consistently encouraged me during moments of doubt. His dedication and mentorship played a key role in shaping both my research and my growth as a young researcher.

I would also like to thank Dr. Amin Nikanjam, whose assistance and feedback were invaluable—especially during the preparation and submission of my research papers.

To my lab mates, thank you for the many moments we shared—you made this journey lighter and more enjoyable. I would especially like to recognize Ahmed Haj Yahmed, who supported and encouraged me during times of self-doubt and helped me make important decisions regarding my Master's journey.

Finally, I would like to acknowledge Pr. Michel Desmarais and Pr. Maxime Lamothe for taking the time to evaluate my thesis.

RÉSUMÉ

Les modèles d'apprentissage automatique (ML) et les grands modèles de langage (LLMs) ont démontré une grande efficacité dans l'automatisation de tâches traditionnellement effectuées manuellement. Leur performance a favorisé leur intégration dans des applications critiques, au point de constituer l'infrastructure de base de nombreux systèmes complexes basés sur l'intelligence artificielle (IA). Pour gérer l'ensemble du cycle de vie de ces modèles, des cadres opérationnels tels que le Machine Learning Operations (MLOps) et le Large Language Model Operations (LLMOps) ont émergé, proposant des outils et des bonnes pratiques adaptés.

Les pipelines MLOps et LLMOps permettent le déploiement continu de modèles améliorés, que ce soit par l'adoption de versions plus performantes ou par l'adaptation des modèles à des environnements dynamiques en constante évolution, via l'entraînement continu (CT) sur des données de production récentes. Ces pratiques visent à renforcer la fiabilité des systèmes basés sur l'IA.

Cependant, malgré les outils et recommandations proposés pour améliorer la robustesse, les pipelines MLOps et LLMOps ne garantissent pas, en eux-mêmes, la fiabilité ni la confiance dans les modèles déployés. Par exemple, le déploiement d'un modèle sous-optimal peut entraîner une dégradation des performances plutôt qu'une amélioration, compromettant ainsi la fiabilité de l'ensemble du pipeline. De tels problèmes de fiabilité peuvent conduire à des échecs coûteux, à une perte de confiance des parties prenantes, voire à des erreurs critiques dans des contextes sensibles.

Dans ce mémoire, nous cherchons à contribuer à l'amélioration de la fiabilité et de la confiance dans les pipelines MLOps et LLMOps. La première partie porte sur le MLOps et vise à renforcer la fiabilité des modèles mis à jour à travers des workflows de CT. Bien que le CT soit conçu pour améliorer les performances des systèmes d'IA, il peut introduire des risques lorsque les données de production sont bruitées ou mal gérées, menant à des régressions catastrophiques ou à une dégradation silencieuse des performances. En pratique, les données de production peuvent présenter un décalage de distribution ou être automatiquement étiquetées avec une faible confiance, les rendant inappropriées pour un apprentissage continu fiable. Pour répondre à ces défis et favoriser des pipelines de CT plus robustes, nous proposons une approche de maintenance fiable reposant sur un mécanisme de filtrage des données entrantes. Cette méthode écarte les instances à faible confiance, potentiellement mal étiquetées, ainsi que les échantillons présentant une forte divergence par rapport à la distribution d'origine. Cette approche permet de sécuriser le processus d'apprentissage continu et de garantir des mises à jour de modèles plus fiables dans le temps.

La seconde partie de ce mémoire s'intéresse aux pipelines LLMOps, en particulier ceux destinés aux tâches de génération de code. Avec l'évolution rapide des grands modèles de langage, de nouvelles versions sont régulièrement publiées, souvent accompagnées de promesses d'améliorations significatives. Toutefois, ces mises à jour peuvent introduire involontairement des régressions, et même les modèles les plus avancés sont susceptibles de générer du code comportant des inefficacités. Cela complique le maintien d'une qualité constante et d'une fiabilité durable au fil des versions.

Pour relever ces défis, nous proposons d'abord une taxonomie des inefficacités couramment observées dans le code généré par les LLMs. Cette taxonomie offre une base structurée pour évaluer systématiquement les sorties des modèles et identifier les défauts récurrents. Sur cette base, nous introduisons ReCatcher, une suite de tests de régression conçue pour détecter à la fois les régressions de capacités et les améliorations entre différentes versions de modèles LLM. ReCatcher contribue ainsi à un processus de déploiement continu plus transparent, plus fiable et mieux informé.

Ensemble, ces contributions visent à renforcer la fiabilité des systèmes basés sur l'IA en traitant des défis clés dans les pipelines MLOps et LLMOps, et en fournissant des solutions concrètes et des perspectives utiles pour un déploiement de modèles plus sûr.

ABSTRACT

Machine Learning (ML) models and Large Language Models (LLMs) have demonstrated strong capabilities in automating tasks traditionally performed manually. Their effectiveness has led to their integration into critical applications, forming the backbone of complex AI-based systems. To manage the full lifecycle of these models, operational frameworks such as Machine Learning Operations (MLOps) and Large Language Model Operations (LLMOps) have emerged, offering tailored tools and best practices. MLOps and LLMOps pipelines enable the continuous deployment of improved models—either by updating to more capable versions or by adapting models to evolving, dynamic environments through Continuous Training (CT) on fresh production data. These practices aim to enhance the dependability of AI-based systems. However, despite offering best practices and tools to support robustness, MLOps and LLMOps pipelines do not inherently guarantee reliability or trustworthiness. For instance, deploying a sub-optimal model may lead to performance degradation instead of improvement, ultimately compromising the reliability of the entire pipeline. Such reliability issues can lead to costly failures, loss of stakeholder trust, and critical errors in high-stakes applications.

In this thesis, we aim to contribute to improving the reliability and trustworthiness of both MLOps and LLMOps pipelines. In the first part, we focus on MLOps and aim to enhance the reliability of models updated through CT workflows. Although CT is designed to improve AI-based system performance, it can also introduce risks when production data is noisy and poorly managed, leading to catastrophic regressions or silent performance degradation. In practice, production data may suffer from distribution drift or be automatically labeled with low confidence, making it unsuitable for reliable CT. To address these challenges and promote more robust CT pipelines, we propose a reliable maintenance approach based on a filtering mechanism for incoming data. This method excludes low-confidence instances, which are likely to be mislabeled, as well as samples that significantly deviate from the original distribution. This approach helps safeguard the CT process and ensures more reliable model updates over time.

The second part of this thesis focuses on LLMOps, particularly pipelines for code generation tasks. With the rapid evolution of large language models, new versions are frequently released, often accompanied by promises of significant improvements. However, such updates can inadvertently introduce regressions, and even the most advanced models may produce code with inefficiencies. This makes it difficult to maintain consistent quality and long-term reliability across model versions.

To address these challenges, we first propose a taxonomy of inefficiencies commonly observed in code generated by LLMs. This taxonomy provides a structured basis for systematically evaluating model outputs and identifying recurring flaws. Building on this foundation, we introduce ReCatcher, a regression testing suite designed to detect both capability regressions and improvements between different LLM versions. ReCatcher thus contributes to a more transparent, trustworthy, and well-informed continuous deployment process for language models.

Together, these contributions aim to strengthen the reliability of AI-based systems by addressing key challenges in MLOps and LLMOps pipelines, while providing concrete solutions and actionable insights for safer model deployment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

AE	Auto-Encoders
AI	Artificial Intelligence
AIOps	Artificial Intelligence for IT Operations
AL	Active Learning
ASTs	Abstract Syntax Trees
AutoML	Automated Machine Learning
CDC	Constraint Disagreement Classifier
CI/CD	Continuous Integration / Continuous Deployment
CL	Continual Learning
CWE	Common Weakness Enumeration
CT	Continuous Training
DL	Deep Learning
FMOps	Foundational Models Operations
KS	Kolmogorov-Smirnov
LaaJ	Large Language Model as a Judge
LLMOps	Large Language Model Operations
LLMs	Large Language Models
ML	Machine Learning
MLOps	Machine Learning Operations
MMD	Maximum Mean Discrepancy
OOD	Out-of-Distribution
SE	Software Engineering
TIES	TrIm, Elect Sign, and Merge
VAE	Variational Auto-Encoder

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

Machine Learning (ML) has fundamentally transformed numerous domains by enabling the automation of tasks traditionally requiring extensive manual effort, thereby enhancing efficiency and scalability across sectors [1, 2]. Early ML systems were predominantly based on hand-crafted, rule-based approaches, wherein domain experts manually encoded logic to reflect specific operational environments [3]. The emergence of Deep Learning (DL) marked a significant paradigm shift by enabling models to autonomously learn complex representations from large-scale data [2].

More recently, generative models, such as Large Language Models (LLMs), have further pushed the boundaries of ML by generating human-like text and realistic images, and solving highly complex tasks without explicit programming [4,5]. As ML-based systems are increasingly adopted in critical applications such as autonomous driving, industrial inspection, and automated code generation [2,6], their deployment introduces new challenges. These systems often integrate multiple models and must meet demanding requirements related to scalability, maintainability, operational reliability, and trustworthiness. In this context, trustworthiness refers to the consistent and predictable performance of a system across varied conditions and over time [7]. Addressing these needs has led to the development of Machine Learning Operations (MLOps), a set of principles, tools, and practices designed to manage the full ML lifecycle, including development, training, deployment, and monitoring in dynamic environments [7–10].

MLOps frameworks reduce manual effort, minimize errors, and enhance collaboration among data scientists, software engineers, and operations teams by standardizing workflows and providing reliable infrastructure. They enable continuous integration and deployment, systematic version control, and reproducibility, all of which are essential for operating ML systems at scale [11,12]. However, the growing adoption of foundation models such as LLMs has introduced new operational demands that exceed the scope of conventional MLOps solutions. Unlike traditional ML models—which are typically lightweight, operate on structured and preprocessed features, and are trained for narrowly defined tasks—LLMs exhibit distinct operational demands. These models are substantially larger, often consisting of billions of parameters, and require specialized hardware and distributed computing environments to support inference and fine-tuning [13]. Moreover, LLMs are driven by natural language prompts rather than fixed input features, shifting the focus from data preprocessing to prompt engineering and management [14]. The stochastic and opaque nature of LLM outputs further complicates tasks such as evaluation, monitoring, and versioning [15]. To address these unique

challenges, the concept of LLMOps has emerged as an extension of MLOps, specifically tailored to the lifecycle and operational requirements of LLM-based systems [16].
By adopting MLOps and LLMOps practices, organizations can automate complex workflows, reduce error-prone manual interventions, and ensure the reliable deployment of performant and up-to-date models [10, 17]. These practices also support model rollback and traceability, facilitate reproducibility, and foster interdisciplinary collaboration by introducing a shared operational vocabulary [11]. Such capabilities are especially important for systems composed of multiple interacting components, where coordination, monitoring, and accountability are critical [18].
Despite these advancements, MLOps and LLMOps pipelines remain vulnerable to silent performance degradation, which can erode the reliability of deployed systems without immediate detection [18, 19]. This degradation can result from various factors, including subtle implementation issues such as mismatched preprocessing logic across different components [20], or from the continuous deployment of models trained on noisy or outdated data. In LLMOps workflows, newer model versions may inadvertently reduce task-specific performance or introduce unexpected behaviors, even when overall model capabilities appear to improve [21]. The consequences of such degradation vary depending on the application. In industrial inspection, an ML model may fail to detect newly emerging defect types due to data distribution shifts. In software development workflows, an LLM-based code generator may produce syntactically correct but inefficient or semantically incorrect code that fails to meet the expectations of developers or downstream tools.

Thesis Objectives

This thesis addresses the reliability challenges posed by the deployment and evolution of ML and LLM-based systems within MLOps and LLMOps pipelines. Its central objective is to enhance the reliability, robustness, and trustworthiness of AI-based systems operating in production environments. Through a combination of empirical analyses, tool development, and automation strategies, the thesis presents practical contributions aimed at mitigating performance degradation and supporting dependable AI lifecycle management.
Specifically, this thesis makes two main contributions:

- *MLOps - Automated maintenance of industrial vision inspection systems* ML is increasingly utilized in industrial visual inspection to support automated quality control, particularly in the context of Industry 4.0 [22]. These systems require frequent updates via CT workflow to remain effective in dynamic production environments. However, without rigorous validation, these updates may introduce performance regressions [19].

This work investigates the root causes of such degradation and develops automated solutions for preserving model reliability across the system lifecycle.

- *LLMOps - Safe model update for LLM-based code generation systems* The adoption of LLMs for automated code generation is accelerating, with generated code increasingly integrated into production systems [23]. However, the complexity and opacity of LLMs, combined with rapid versioning cycles, introduce significant risks related to behavioral regressions and quality degradation. This thesis proposes tools and methodologies to systematically evaluate, test, and manage LLM updates in code generation pipelines, ensuring safe and reliable deployment.

Ensuring reliability within both MLOps and LLMOps pipelines is critical, as failures can lead to substantial financial loss, operational disruption, or safety hazards [24]. These challenges highlight the need for rigorous techniques to prevent performance degradation as models evolve within production environments.

In the following sections, we first examine MLOps by exploring the use of ML in industrial visual inspection systems and identifying the reliability challenges that arise throughout their lifecycle, such as data drift and automated labeling. We then turn to LLMOps, focusing on the adoption of LLMs for code generation and the reliability issues they pose, including inefficient outputs, challenges in evaluation, and the absence of structured testing and regression techniques.

1.1 MLOps: Reliability Challenges in Visual Inspection Systems

With the advent of Industry 4.0, ML models are increasingly being deployed across a wide range of industrial applications, including visual inspection systems [22]. These systems often involve complex pipelines in which multiple ML models interact to perform tasks such as defect detection and quality assessment [7]. However, maintaining the reliability of these systems in dynamic production environments remains a significant challenge, as industrial settings are frequently subject to change—for example, the introduction of new products or the emergence of novel defect types [25].

To address these evolving conditions, MLOps guidelines recommend the use of CT workflows that leverage newly collected production data. While CT can support ongoing adaptation and help maintain model performance over time, its effectiveness depends heavily on the integration of robust data validation mechanisms [26]. These mechanisms act as quality control gates, ensuring that incoming data is suitable for retraining. Data validation typically involves checks for distributional consistency and the correctness of automatically generated labels [27]. When such checks are omitted or inadequately implemented, CT workflows can result in performance degradation [19]. The following subsections explore two key reliability risks in CT workflows: data drift, where shifts in data distribution affect the model's ability to generalize, and automated labeling, where incorrect annotations can silently introduce noise into the training data.

1.1.1 Data Drift

Data Drift refers to the phenomena where the distribution of production data, gradually or drastically, deviates from originally training data [28]. CT workflows that incorporate drifted data can pose a significant risk to the reliability of MLOps pipelines—namely, catastrophic forgetting, where the model becomes overly tuned to recent data and loses its ability to generalize from previously learned patterns. Several techniques have been proposed to detect and manage data drift in MLOps pipelines [19]. In computer vision applications, a common approach involves encoding input images into feature representations before applying classifiers to detect significantly different samples indicative of drift [29, 30]. Autoencoders (AEs) are commonly used for this purpose because of their ability to capture semantic features. However, they can overlook critical structural or spatial information, which may result in incomplete or insufficient drift detection.

1.1.2 Automated Labeling

Supervised learning models require labeled data to support CT workflows. In automated MLOps pipelines, relying on manual labeling for the large volumes of incoming production data is impractical and undermines scalability [31]. As a result, automated labeling techniques are commonly employed to annotate production data without human intervention. These approaches often rely on pre-trained models, ensemble predictions, or weak supervision strategies to generate labels for new data [32–34]. While automated labeling plays a crucial role in enabling end-to-end retraining pipelines, it introduces a new layer of risk. Inaccurate or noisy labels can propagate errors into the training loop, especially if there is no verification mechanism in place. Retraining on incorrectly labeled data can significantly degrade model performance and result in catastrophic forgetting; a phenomenon in which the model adapts to erroneous patterns while losing previously learned, valid knowledge [35]. This issue poses a critical threat to the reliability of MLOps pipelines.

1.2 LLMOps: Reliability Challenges in LLM-based Code Generation Systems

LLMs have demonstrated remarkable capabilities in various Software Engineering (SE) tasks, particularly in automated code generation [36]. These models can generate code in multiple programming languages, including Python, Java, and C/C++ [37], and are increasingly adopted in production codebases and multi-agent development workflows [38, 39]. Despite the promising capabilities of LLMs, code generated by these models often exhibits inefficiencies that compromise essential software quality attributes, including correctness, readability, maintainability, and performance [40–43]. These shortcomings are particularly concerning when LLM-generated code is deployed at scale, where reliability, trustworthiness, and long-term maintainability are paramount. Ensuring reliable code generation within LLMOps pipelines necessitates the development and application of comprehensive evaluation methods. However, current evaluation practices are often limited in scope and fail to capture the full spectrum of quality-related concerns in generated code. The following subsections explore this gap in greater detail.

1.2.1 Limitations of Current Evaluation Practices

To evaluate the code generation capabilities of LLMs, researchers and practitioners primarily rely on automated evaluation metrics such as CodeBLEU, which measures syntactic and semantic similarity between generated and reference code, and Pass@k, which evaluates functional correctness by testing whether a generated solution passes unit tests within k attempts [23]. These metrics offer a high-level overview of LLM performance and enable comparative analysis across different models.

However, these evaluation methods have notable limitations. In particular, they emphasize functional correctness while largely neglecting non-functional attributes such as code readability, maintainability, and runtime performance [44]. As a consequence, LLMs may achieve high Pass@k scores while producing suboptimal or difficult-to-read code. When such code is integrated into production systems, it can accumulate technical debt, introduce hidden performance bottlenecks, and negatively impact long-term software quality and maintainability.

1.2.2 Emerging Need for Systematic Testing in LLMOps

Unlike traditional software development, where rigorous testing and debugging cycles are applied, LLM-generated code typically lacks systematic validation mechanisms beyond pass@k-style evaluations [23]. Current evaluation methods do not account for regression testing - an essential process for ensuring that model updates do not degrade previously correct behaviors or introduce new inefficiencies. The absence of such structured testing increases the risk of silent regressions, where newer model versions preserve functional correctness but produce suboptimal, inefficient, or unnecessarily complex code [21]. Furthermore, as LLMs become increasingly integrated into agent-based development workflows and production environments, the need for reliable mechanisms to manage model updates, enforce consistency checks, and apply quality gating becomes urgent. A well-structured code generation LLMOps pipeline should support the detection of LLM regressions not only in terms of functional correctness but also with respect to non-functional properties such as code efficiency, readability, and maintainability [45]. In the absence of such safeguards, LLM-assisted development risks introducing low-quality artifacts into production codebase, potentially undermining confidence in automated tools and increasing long-term maintenance costs.

1.3 Thesis Statement

Despite the increasing adoption of MLOps and LLMOps pipelines to support the maintenance of AI-based systems in real-world environments, current practices often fall short in ensuring reliability and trustworthiness in production settings. Specifically, existing approaches face two major limitations: (i) in MLOps, there is insufficient support for robust CT workflow on fresh production data, which may be affected by data drift or automated labeling errors; and (ii) in LLMOps, current methods lack accurate evaluation and comparison mechanisms for assessing the capabilities of large language models in code generation tasks.
These shortcomings undermine the dependability of AI systems, increasing the risk of undetected silent regressions in critical production workflows.
This thesis addresses these gaps by proposing:

- A robust two-stage data filtering approach to support reliable MLOps of DL models by filtering out drifted and incorrectly labeled data;
- Support for LLMOps through (i) a taxonomy of code inefficiencies to enable structured evaluation of LLM-generated code, and (ii) a regression testing framework to systematically compare LLMs across multiple quality dimensions and ensure reliable updates.

Together, these contributions advance the state of MLOps and LLMOps by promoting greater reliability, efficiency, and maintainability of AI-based systems in real-world production environments.

1.4 Thesis Overview

In this section, we provide an overview of the key studies that form the foundation of this thesis.

- *MLOps* - We develop a robust CT workflow to support the reliable maintenance of DL visual inspection systems. This approach includes a two-stage data filtering process: the initial stage filters out low-confidence predictions and the second stage uses variational auto-encoders (VAE) and histograms-based embeddings to detect and reject drifted inputs. Fine-tuning is subsequently conducted on the filtered data, with validation performed on both the production and original datasets to detect and mitigate catastrophic forgetting. Evaluations on real-world inspection tasks (popsicle stick prints and glass bottles) show that the approach retains less than 9% of erroneous self-labeled data and improves production performance by up to 14% without compromising validation accuracy.

- *LLMOps* - We construct a comprehensive taxonomy of inefficiencies in LLM-generated code based on a manual analysis of 492 code samples generated by CodeLlama, DeepSeek-Coder, and CodeGemma on the HumanEval+ benchmark. The taxonomy categorizes inefficiencies into five major dimensions: *General Logic*, *Performance*, *Readability*, *Maintainability*, and *Errors*, with 19 subcategories. Validation through a survey of 58 LLM practitioners and researchers confirms the taxonomy's completeness, relevance, and real-world applicability. Our qualitative findings indicate that inefficiencies are diverse and interconnected, affecting multiple aspects of code quality, with *logic* and *performance-related* inefficiencies being the most frequent and often co-occur while impacting overall code quality.

- *LLMOps* - We propose *ReCatcher*, the first LLMs regression testing framework for Python code generation. *ReCatcher* systematically assesses regression between two models (i.e., a currently used model and a candidate for update) using proven software testing tools, ensuring a reliable assessment. We use *ReCatcher* to evaluate regression across three model update scenarios — fine-tuning, merging, and model release —on three state-of-the-art models, i.e., CodeLlama, DeepSeek-Coder, and GPT-4o. Our findings reveal that fine-tuning on datasets from different programming languages increases syntax errors by up to 12%. While merging with general-purpose LLMs (non-optimized for code) can introduce regression across various aspects, such as logical correctness, by up to 18%. Additionally, GPT-4o exhibits regressions of up to 50% in handling missing imports compared to GPT-3.5-turbo, while GPT-4o-mini shows performance regression in execution time compared to GPT-4o by up to 80%.

1.5 Thesis Contribution

This thesis makes the following key contributions toward enhancing the reliability and robustness of MLOps and LLMOps pipelines in production:

- *Trimming the Risk: Towards Reliable Continuous Training for Deep Learning Inspection Systems*

We develop a CT-based maintenance approach that minimizes the risks of retraining on drifted or incorrectly labeled data. The method employs a two-stage filtering pipeline: confidence-based rejection of low-certainty predictions and distribution-based drift filter using VAE and histogram embeddings. Applied to real-world industrial inspection systems under critical scenarios, this approach retains less than 9% of mislabeled data and improves production accuracy by up to 14% while preserving model stability on original validation sets.

- *A Taxonomy of Inefficiencies in LLM-Generated Code*

  We present a validated taxonomy that categorizes inefficiencies in LLM-generated code across five main dimensions and 19 subcategories. Based on manual analysis of 492 code samples and supported by feedback from 58 practitioners, the taxonomy offers a structured framework for evaluating non-functional aspects of LLM outputs, including performance, maintainability, and readability. Our qualitative findings indicate that inefficiencies are diverse and interconnected, affecting multiple aspects of code quality, and that logic and performance-related issues are the most frequent and interdependent inefficiencies.

- *ReCatcher: Towards LLMs Regression Testing for Code Generation*

  We propose *ReCatcher*, the first LLMs regression testing framework for Python code generation. *ReCatcher* systematically assesses regression between two models (i.e., a currently used model and a candidate for update) using proven software testing tools, ensuring a reliable assessment. We use *ReCatcher* to evaluate regression across three update scenarios for LLMs — fine-tuning, merging, and model release —on three state-of-the-art models, i.e., CodeLlama, DeepSeek-Coder, and GPT-4o. Our findings reveal that regression most frequently affects *Errors* (syntax error and missing declaration/import), *Performance*, and *Logical correctness* of generated code. However other code quality aspects such as *Maintainability* and *Readability* are overall stable across updates.

1.6 Thesis Organization

The remainder of the thesis is organized as follows:

- Chapter 1: Provides an overview of the preliminary concepts and knowledge required for comprehending the subsequent sections of the thesis.

- Chapter 3: Reviews the relevant literature on MLOps and LLMOps, with a focus on existing challenges in CT, and inefficiencies in LLM-generated code.

- Chapter 4: Presents a robust CT-based maintenance approach for DL visual inspection systems, aiming to ensure reliability within MLOps pipelines.
- Chapter 5: Advances LLMOps by introducing a taxonomy of inefficiencies in LLM-generated code.
- Chapter 6: Proposes ReCatcher, an LLM regression testing framework for code generation.
- Chapter 7: Presents the conclusion of the thesis and discusses limitations and future work.

CHAPTER 2 BACKGROUND

Chapter Overview. This chapter introduces the foundational concepts of MLOps (2.1), highlighting key components such as continuous training (2.1.1), automated labeling (2.1.2), and data drift (2.1.3). We also present the emerging field of LLMOps (2.1.4), which extends MLOps practices to LLMs. In the second part of the chapter, we provide background on the two primary application domains explored in this thesis: industrial visual inspection (2.2) and LLMs with a focus on code generation (2.3).

2.1 Machine Learning Operations (MLOps)

MLOps is a set of principles, tools and best practices proposed to optimize the development, deployment and maintenance of ML models in production environments [9, 10]. Inspired by DevOps, MLOps aims to bridge the gap and ensure seamless collaboration between data scientists, software engineers and IT operation team and ensure efficient model management through ML life cycle [8]. MLOps is defined by 7 key principles:

- *Continuous Integration and Continuous Deployment (CI/CD)*: automates the building, testing, and deployment of ML pipelines, ensuring rapid feedback and iterative development.
- *Workflow orchestration*: manages task dependencies using directed acyclic graphs for efficient pipeline execution.
- *Reproducibility and Versioning*: guarantee traceability and repeatability of experiments by tracking data, code, and model artifacts.
- *Collaboration*: ensures alignment and shared responsibilities across interdisciplinary teams.
- *Continuous training and evaluation*: keeps models up-to-date by periodically retraining and validating them on fresh data.
- *Continuous monitoring* tracks both model performance (e.g., accuracy, drift) and system metrics (e.g., latency, resource usage) in production.
- *Feedback loops* incorporates insights from production data and user behavior to inform iterative model improvement and lifecycle governance.

MLOps practices can be adopted at different levels of maturity, depending on the extent of automation across different stages of the ML lifecycle. Several frameworks have been proposed by industry and academia to characterize these maturity levels [16]. For instance Google proposed a model with 3 maturity levels [46]:

- *Level 0*: represents a fully manual workflow.
- *Level 1*: automates the ML pipeline to support CT workflow. This level includes mechanisms for automated data validation, model validation, and triggers for launching training pipelines.
- *Level 2*: proposes a fully automated CI/CD pipeline. It supports continuous integration of new code and pipeline components through automating testing building and packaging and continuous delivery ensuring that new models and pipeline changes are automatically deployed.

While MLOps is increasingly adopted in practice, it is often confused with related but distinct concepts such as AutoML and AIOps. AutoML (Automated ML), refers to the automation of model development tasks such as feature engineering, model selection and hyperparameters tuning [47]. In contrast to MLOps which primary addresses operationalization and life cycle of ML workflows. Regarding AIOps (Artificial Intelligence Operations), it stands for applying AI to automate and enhance IT operations such as anomaly detection, root cause analysis and incident prediction [48].
In the following, we delve into key MLOps concepts and associated risks, including continuous training, automated labeling, and data drift. We then transition to LLMOps, an emerging subfield of MLOps specifically tailored to the operationalization of LLMs.

2.1.1 Continuous Training (CT)

Continuous Training (CT) is an incremental learning practice that involves updating a model with new data or tasks overtime, without forgetting previously learned knowledge [49]. CT typically involves automatically launching retraining workflow for ML models to adapt to changing environments on a regular basis [50]. There are different approaches implementing the CT that can be categorized based on the trigger: periodic retraining, performance-based retraining, or data-driven retraining and on demand retraining [50]. When it comes to the retraining data used, two main approaches emerge: training the model from scratch on all available data [51] or transfer learning, in which a previously trained model is fine-tuned to adapt to new data [52]. Online learning, also known as stream learning, offers a cost-effective alternative to offline CT by leveraging algorithms that incrementally learn from new data streams without requiring explicit triggers [53]. CT is often confused with Continual Learning (CL). However, CL is a broader research practice area focused on enabling models to accumulate knowledge across sequential tasks without catastrophic forgetting. Unlike CT, CL does not require automation, is not necessarily tied to production environments, and is not inherently part of MLOps workflows [54].

2.1.2 Automated Labeling

Manual data labeling is often criticized as laborious and time-consuming step in ML engineering pipeline, and it can also become costly if it necessitates subject-matter experts [31]. To reduce costs, ML models can be used alongside humans in a semi-supervised approach [32], where the model proposes annotation suggestions to humans to accelerate the labeling process. Similarly, Active Learning (AL) is employed to optimize labeling efforts by selecting only the most informative datapoints [33]. Despite reducing labor costs, these human-in-the-loop approaches cannot be fully automated. Alternatively, the Snorkel framework enables the programmatic creation of labeling functions to auto-label most of the data based on heuristics and domain knowledge [34]. However, these functions are not trivial to design, are prone to errors, and typically cover only a subset of the data—requiring humans to remain partially in the loop to ensure complete dataset labeling. Another straightforward option is to use an ML model trained on labeled data for labeling unseen data [55]. Yet, model errors make this method prone to producing corrupted labels. For this reason, calibrated confidence and uncertainty components are required to avoid relying on the supervised model when its predictions are uncertain [56].

2.1.3 Data Drift

Data Drift refers to changes in production data distribution overtime, which can be gradual, sudden or seasonal [57], affecting the statistical properties or characteristics of the model's input data [28]. The dynamic nature of production environment causes data to shift from the original data distribution [58]. The shift may reveal the emergence of new defects (uncovered by the training data) [59] or the rise of a new environment [60], which may include changes such as new backgrounds, blur, noise, illumination, occlusion [61, 62]. Data drift adversely affects model performance, though does not necessary invalidate its learned patterns and inductive bias [63], contrary to concept drift which describes data evolution, invalidating the current model [64]. There are closely related concepts to data drift that may cause confusion. For example Out-of-Distribution (OOD), refers to to individual data points that fall outside of the distribution [65], whereas data drift concerns a shift in the overall data distribution. Similarly, training-serving skew describes a mismatch between the training data and the data observed during inference—even if the production data distribution remains unchanged [66].

2.1.4 Large Language Models Operations (LLMOps)

LLMOps is an emerging discipline that extends the traditional MLOps to address unique challenges associated with deployment, monitoring, and maintenance of LLM-based applications [45]. While MLOps focuses on automating and streamlining the lifecycle of machine learning models, LLMOps introduces specialized practices tailored to the complexities of LLMs [16]. LLMs are large models, that consist of hundreds of billions of parameters, requires substantial computational power for adapting (such as fine-tuning and inference) and costs are dominated by GPU-infrastructure. LLMOps ensures efficient resource utilization, cost efficiency and scalability [16]. Additionally, LLMs have specialized adaptation techniques to tailor their behavior for specific applications different from standard ML model training. One such technique is prompt engineering, which involves crafting specific prompts to guide the model's responses effectively. Also fine-tuning with domain-specific data can be used. These adaptation methods are integral to LLMOps, which provides structured methodologies to support their implementation [16]. Moreover, the textual output of LLMs makes evaluation and monitoring harder posing unique challenges due to the inherent subjectivity and variability of language [67]. Finally, LLMs present some unique risks such as hallucination where the output appears plausible but is factually incorrect or nonsensical [68]. LLMOps extends MLOps by addressing conventional ML lifecycle challenges and other specific to LLMs [16].
LLMOps may be confused with FMOps (Foundational Models Operations) which focuses on best practices and guidelines for developing, training, and evaluating foundation models. In contrast, LLMOps specifically addresses the operationalization of applications powered by large language models [69].

2.2 Industrial Visual Inspection

Industrial inspection is a quality control process used across industries to examine manufacturing products and equipment for visual appearance aiming to identify both functional and cosmetic defects [70, 71]. It has been used across many industries such as automotive, manufacturing, robotics, electronics and aerospace [70, 72–74]. Visual inspection can be performed using several approaches. Traditionally, it has been conducted by human inspectors who identify defects using the naked eye, sometimes aided by tools such as flashlights or magnifying glasses [71, 75]. However, relying on human inspectors can be costly and inefficient, as human inspectors demand repetitive observation tasks that can lead to fatigue, reduced attention, and variable performance across different inspectors [75]. These limitations result in inconsistent detection rates, reduced productivity, and increased production costs [71]. To address these limitations and leverage technological advancements, automated visual inspection has emerged as an alternative. In automated visual inspection, manufacturing lines are equipped with cameras strategically positioned to capture images of products moving along the production line. Captured images are then automatically processed using computer vision algorithms [76]. Initially, defects are detected using rule-based pattern recognition approaches, where predefined rules by experts are used to identify deviations in the visual characteristics of a product [3]. Although rule-based inspection systems significantly reduce manual effort, they suffer from several limitations [77]. They require frequent manual updates to adapt to changes in industrial conditions or product design, and developing comprehensive rules can be particularly challenging for complex products with subtle defects. More recently, the advancements in ML and DL techniques have effectively enhanced the effectiveness of automated visual inspection [2]. ML-based visual inspection systems have proven highly efficient in detecting and classifying defects due to their ability to autonomously learn complex and subtle patterns from large datasets, thus reducing the dependency on manually crafted inspection rules. With the advent of Industry 4.0, MLOps practices have been integrated into industrial inspection workflows to automatically deploy, monitor, and continuously adapt to the dynamic environment [22]. In the literature, several approaches have been proposed for visual inspection tasks. Supervised methods, including object detection and semantic segmentation, are widely adopted for accurately localizing defects within images [70, 78, 79]. Alternatively, unsupervised methods, notably one-class classification, have gained popularity for scenarios characterized by limited or unavailable labeled defect data [80]. To facilitate consistent evaluation and comparison among these approaches, standardized benchmarks such as the MVTec Anomaly Detection dataset have been established [81].

2.3 Large Language Models (LLMs)

LLMs are advanced AI systems designed to understand, process, and generate human-like text. As generative models, LLMs take a given input, a prompt, and generate a text as output [82]. While their output is text, the applications of LLMs extend far beyond conversational agents. They have demonstrated efficiency across various domains such as finance, robotics, and software engineering [83–85]. LLMs are trained on massive amounts of textual data, resulting in large models that may reach hundreds of billions of parameters [82]. Despite their powerful capabilities, LLMs are not devoid of limitations. One notable issue is hallucinations, where the model generates information that is incorrect or nonsensical [86]. Additionally, LLMs are prone to biases present in their training data, which can impact the quality and fairness of their outputs [87]. In the following, we explore the use of LLMs in code generation and discuss adaptation techniques such as merging and fine-tuning that enhance the capabilities of LLMs.

2.3.1 LLMs for Code Generation

LLMs have demonstrated their capability to generate code across various programming languages, including Python, Java, and C/C++ [37]. This thesis explores the use of different models for code generation, and in the following section, we provide a detailed description of each of the models utilized in our study.

CodeLlama

CodeLlama is an open-source family of models based on Llama 2 [88] developed by Meta for code-related tasks such as completion and infilling. It includes the foundation model (CodeLlama), a Python-specialized model (CodeLlama-Python), and an instruction-following model (CodeLlama-Instruct), available in different sizes (7B, 13B, 34B, and 70B parameters). CodeLlama is trained on a near-deduplicated dataset of publicly available code, it incorporates a small fraction of natural language data related to programming to enhance comprehension [89].

DeepSeek-Coder

DeepSeek-Coder provides a range of open-source models trained from scratch on 2 trillion tokens across 87 programming languages. The pretraining data is organized at the repository level to enhance model capability [90]. The models employ two key techniques: fill-in-the-middle and next-token prediction. Just like CodeLlama, DeepSeek-Coder is also available in different sizes, ranging from 1.3B to 33B parameters, with both foundational and instruction-tuned versions. These models leverage a 16K context window to improve code generation and infilling [90].

CodeGemma

Code Gemma is a collection of specialized open-code models based on Google DeepMind's Gemma models [91], trained on 500 to 1000 billion tokens, primarily for code-related tasks. The CodeGemma family includes: a 7B pre-trained model, a 7B instruction-tuned model, and a specialized 2B model designed for code infilling and open-ended generation. The 2B model is trained entirely on code, while the 7B models are trained on a mix of 80% code and 20% natural language, sourced from deduplicated publicly available code repositories. These models are trained using a fill-in-the-middle task to enhance their coding capabilities [92].

GPT-Family

OpenAI models have demonstrated strong code generation capability [93]. GPT-3.5, optimized for chat applications, includes the GPT-3.5-turbo model, which is fine-tuned for enhanced language comprehension and text generation capabilities. The GPT-4 family introduces GPT-4o, a multimodal model capable of processing and generating text, images, and audio. GPT-4o-mini is a smaller, more cost-efficient version of GPT-4o, offering similar capabilities at a reduced computational cost [94–96].

2.3.2 LLMs Adaptation Techniques

Model adaptation tailors LLMs for specific tasks or domains by leveraging their pre-existing capabilities, reducing the need for extensive training from scratch [97]. In this thesis, we study the impact of merging and fine-tuning techniques.

Merging

Model merging (or fusing) combines parameters from multiple LLMs to create a new model with integrated capabilities without requiring access to the original training data [98]. This technique preserves and enhances the strengths of each merged model. TIES (TrIm, Elect Sign, and Merge) is a notable merging method that: (i) resets minimally changed parameters during fine-tuning, (ii) resolves sign conflicts among parameters, and (iii) merges only aligned parameters to retain their strengths [99]. Merging has been applied for different applications [100] such as SE tasks [98].

Fine-tuning

Fine-tuning adjusts LLMs' parameters through additional training on a smaller, domain-specific dataset, enhancing its capability to addressing domain-specific tasks [101]. Fine-tuning can be supervised and unsupervised, depending on data availability. Instruction fine-tuning further refines models by exposing them to structured prompts and expected outputs [102]. Fine-tuning has demonstrated its effectiveness in many domains [102]. For example, in SE, LLMs have been fine-tuned to improve automated code review [103] and code generation [104].

2.4 Chapter Summary

In this chapter, we introduced the foundational concepts of MLOps, with a particular focus on LLMOps, and provided an overview of the two key application domains studied in this thesis: industrial visual inspection and code generation using LLMs. In the next chapter, we review the literature on existing approaches and techniques aimed at ensuring the reliability of MLOps and LLMOps pipelines, with a focus on challenges related to LLM-based code generation.

CHAPTER 3 RELATED WORK

Chapter Overview. This chapter reviews the state of the art in ensuring the reliability of AI-based systems, focusing on two key applications: (i) MLOps pipelines for visual inspection, and (ii) LLMOps pipelines for code generation.

3.1 Conventional MLOps challenges: Data drift and Continuous training

Over the last few years, with the wide adoption of ML models in dynamic environments, extensive research has been conducted to ensure the reliability of MLOps pipelines. In this section, we first review techniques for detecting data drift and then explore how MLOps pipelines can automatically adapt to such drift, with a particular focus on continuous training. We follow by examining how MLOps practices are applied in the context of industrial inspection systems to ensure reliability and robustness.

3.1.1 Data Drift Detection

Many drift detection approaches are proposed mainly employing either model certainty or data distribution as indicators. Confidence metrics are often leveraged, operating on the assumption that higher confidence implies more accurate predictions, indicating alignment with the known environment settings. For instance, [105] compares confidence scores, i.e., softmax logits, to pre-defined thresholds, and [106] conducts tests that compare original confidence distributions (i.e., softmax logits) with new distribution logits. Similarly, [28] performs two tests on two distributions: confidence of the predicted class and confidence for all other classes. To use calibrated confidences, [107] opts for Constraint Disagreement Classifiers (CDCs) to estimate the aggregated cross-entropy, i.e., confidence, and [108, 109] leverage confidence calibration by data augmenting using generative models. These approaches use model confidence scores as a proxy to model familiarity of data indicating drift, while other approaches use data directly.

To identify potential drifts in imaging data, dimensionality reduction is necessary since distributional analysis is more effective with a low-dimensional representation. Generative models such as AEs and VAEs [29, 30] can be used to derive latent features, thereby encoding the most relevant factors for reconstructing images. In addition to exploring geometric distances within the dimensionally-reduced space, alternative strategies opt for statistical tests such as Kolmogorov-Smirnov (KS), Chi-square, or Maximum Mean Discrepancy (MMD) [29], as well as two-sample testing [30]. [110] performs statistical testing on the reconstruction loss from VAE aggregated with a domain variable. Nevertheless, statistical tests often assume an underlying data distribution and can be sensitive to sample size and number of dimensions. [111] also combines loss with domain variables, but feeds them into an OCC model to capture the pertinent in-distribution characteristics, allowing the systematic elimination of drifted inputs.

3.2 Continuous Training

Continuous Training (CT) refers to the automatic retraining of machine learning models as new production data becomes available. Designing an effective CT strategy involves several critical decisions [112]. One major consideration is when to retrain the models. While periodic retraining is simple to implement, it may result in unnecessary resource usage [113]. In contrast, trigger-based retraining aims to launch training only when it is needed, based on indicators such as distribution deviation. Although more efficient, it introduces the challenge of defining accurate and reliable triggers, which can be difficult in practice [114]. Another key aspect of a CT strategy is data selection. One approach is to use all historical data during retraining to preserve learned patterns and maintain model stability [112]. To reduce the computation cost associated, other strategies to select the training data include fixed and adaptive window sizes, or the use of representative samples [115]. While these approaches can be cost-effective, they add extra complexity. CT can also be implemented in offline or online modes. Offline CT typically involves retraining in batches, whereas online or incremental CT updates the model continuously as new data arrives. Online approaches offer adaptability and faster integration of new data but can be more complex to manage and validate [116]. Finally, transfer learning and continual learning are often leveraged in CT settings. Transfer learning involves reusing pre-trained weights and fine-tuning them on new data, whereas continual learning aims to adapt the model incrementally without catastrophic forgetting [117, 118].
Each CT strategy has trade-offs in terms of performance stability, computational cost, and implementation complexity. The ideal strategy depends on the dynamics of the production environment and the availability of computational resources.

3.3 MLOps Pipelines for Visual Inspection Systems

Ensuring the reliability of visual inspection systems in dynamic industrial environments. As manufacturing processes evolve, data distributions can introduce drift which poses risks to the performance of deployed models. Various solutions have been proposed to detect and adapt to such drifts, often leveraging continuous training. In this section, we review existing approaches adopted in industry.
Samsung proposed an MLOps architecture for defect inspection, covering tasks such as

scratch detection, dent recognition, and missing part identification [7]. Their pipeline is deployed across multiple edge devices and relies heavily on human annotators for labeling incoming data before model retraining. While this human-in-the-loop setup mitigates the risk of incorporating drifted data into the CT process, it lacks automation and scalability in dynamic environments. Additionally, Fabforce [119] has proposed an unsupervised monitoring approach to detect drift by analyzing the latent representation of the classifier embeddings using one-class classification methods. Once a drift is detected, the system flags the input for manual investigation [120]. This approach, while effective at identifying drift, does not automate model adaptation to the dynamic environment as it requires manual intervention and lacks an integrated continuous training mechanism. To enable the automated MLOps pipeline adaptability, a continual learning approach is proposed [121]. The proposed approach starts by detecting new defect types (drift), which are then manually labeled and incorporated into the training set. CT is then triggered using the enriched dataset to ensure adaptability to the evolving defect patterns. Although this approach improves adaptability, it depends on manual annotation to launch CT cycles. Finally, given the scarcity of labeled data, active learning using weak partially labeled unsupervised data may be used [122]. Although this approach addresses the labeling bottleneck,it does not address the risk of data drift.

3.4 Identified Gap and Proposed Direction

Industrial production environments are inherently dynamic, leading to the continuous emergence of drifted data. This poses significant challenges for MLOps pipelines, particularly when automating CT workflows. Without appropriate safeguards, retraining models on drifted or mislabeled data can result in performance degradation, ultimately compromising the reliability and robustness of the deployed system. Although drift detection and adaptation to changing environments have been extensively studied in the research community and are beginning to gain traction among practitioners, current industrial solutions often remain manual or only partially automated. Most existing approaches rely either on confidence-based uncertainty measures or on distribution-based drift detection, but rarely combine both in a cohesive pipeline [119]. To address this gap, we propose a fully automated model maintenance approach that integrates both uncertainty and distributional signals to improve reliability. Specifically, our method introduces a two-stage filtering pipeline:

- *Confidence-based filtering* is used as a first step to eliminate low-confidence predictions, thereby reducing the risk of incorporating mislabeled data. This step assumes that confident predictions are more likely to be correct and self-aware unreliable predictions can be discarded.

- *Distribution-based filtering* is then applied to identify inputs that are substantially drifted from the in-distribution data. Unlike previous work that relies on loss or reconstruction error, we propose using rich image embeddings that combine both semantic and structural information to capture subtle and complex shifts.

We also propose a novel embedding strategy that merges latent representations (from a VAE encoding) with histogram-based pixel profiles. While latent embeddings provide high-level semantic features, the histogram encoding captures fine-grained, low-level pixel characteristics—ensuring that both semantic and structural drifts are accounted for. Furthermore, to ensure real-world applicability, we applied our approach to two industrial visual inspection use cases: identifying defects in glass bottles and detecting printing anomalies on popsicle sticks.

3.5 LLMOps for Code Generation: Output Validation, Evaluation and Testing Challenges

Automated code generation, a long-standing dream in SE, has begun to materialize with the emergence of LLMs capable of generating code across many programming languages such as Python, Java, Rust, C/C++ [37]. In the following we will examine the lack of proper validation in LLM-based application for code generation. We will review the inefficiencies in LLM-generated code, followed by the existing evaluation methods and their limits. Finally we will highlight the growing need for regression testing.

3.5.1 LLM-generated Code Inefficiencies

Prior research has explored various aspects of LLM-generated code, identifying multiple inefficiencies, including those related to performance and readability. Recent studies have systematically assessed inefficiencies in LLM-generated code—spanning readability, maintainability, and performance—by leveraging automated tools such as static analysis, code execution frameworks, and quantitative metrics like pass@k, code complexity, and lines of code. In terms of readability, LLM-generated code often deviates from standard coding best practices and conventions, leading to reduced clarity and long-term maintainability inefficiencies [40, 41]. Although LLMs can generate shorter solutions for complex tasks, their generated code often exhibits higher structure complexity, making them more difficult to read, maintain, and debug [123]. Additionally, LLM-generated code is susceptible to various code smells, which may harden further maintenance [124]. Maintainability issues are also reflected by the occurrence of *improper documentation, unused or undefined variables, and security vulnerabilities* [43]. LLM-generated code further raises safety concerns, as they are prone to critical vulnerabilities, including several from the Common Weakness Enumeration (CWE) Top 25, reinforcing concerns that LLM-generated code may introduce significant security flaws if not carefully reviewed [125]. Performance inefficiencies present an additional challenge. Although LLMs can generate functionally correct solutions, the generated code often lacks efficiency, leading to suboptimal execution times and increased resource consumption [42, 126].

Other related works opted for manual analysis for deeper investigation to identify existing inefficiencies. Existing studies focusing exclusively on buggy code limiting their scope to correctness issues. Tambon et al. [127] analyzed buggy open-source LLM-generated code and introduced a taxonomy of bug patterns, while Dou et al. [123] categorized the root causes of common bugs. A gap remains in comprehensively studying inefficiencies beyond correctness, as even functionally accurate code may suffer from performance degradation, maintainability issues, or security vulnerabilities [42].

3.5.2 LLMs for Code Generation: Evaluation

To evaluate LLM-generated code, two components are essential: (i) a benchmark to generate and test code, and (ii) appropriate metrics for assessment. This section first reviews existing benchmarks used in LLM evaluation and then discusses the main classes of evaluation metrics.

Evaluation Benchmarks for LLMs in Code Generation

LLMs for code generation are assessed across a variety of benchmarks, each targeting different levels of abstraction. For example, HumanEval [38] and MBPP [128] focus on programmatic tasks, each accompanied by unit tests. Enhanced versions like HumanEvalPlus [37] extend these with additional test cases to improve robustness. To assess model capabilities at the class level, ClassEval was proposed [129], while CoderEval targets non-standalone evaluation tasks [130]. Beyond algorithmic problem-solving, BigCodeBench [131] introduces a more diverse set of tasks drawn from domains such as web development, machine learning, and data analysis. Despite this diversity, all these benchmarks share a common focus: assessing functional correctness by providing predefined tasks and corresponding unit tests. Recent studies also introduce benchmarks that assess additional dimensions. For instance, Mercury [132] focuses on the ability to solve competitive programming problems from LeetCode, and EFFICHECH [133] proposes a set of tasks that are specifically designed to surface performance inefficiencies in code.

LLMs Evaluation Metrics for Code Generation

Evaluation metrics for LLM-generated code fall into three main categories: manual evaluation, similarity-based metrics and execution-based metrics [134]. Regarding manual evaluation, it involves experts who assess generated code quality via blind peer reviews, typically based on criteria such as correctness, clarity, or overall quality. While these assessments are detailed, they are inherently not scalable and subjective, limiting their use in large-scale studies [134, 135]. While similarity-based metrics compare generated code to a reference implementation [134]. Common examples include BLEU, ROUGE, and Exact Match derived from natural language processing [23]. However, these metrics treat code as plain text, ignoring its structural and semantic properties. To address this limitation, more specialized metrics such as CodeBLEU [136] have been introduced to incorporate code structure and semantics by leveraging features like ASTs (Abstract Syntax Trees), data-flow, and token alignment. While these metrics are fully automated and scalable, they often fail to account for logically equivalent implementations that differ syntactically. Execution-based metrics evaluate correctness by executing the code. For instance, pass@k measures the percentage of successful test passes within the top-k generated completions. Compilation or interpretation success rates are also used [2]. These metrics do not require reference code and are generally more reliable for assessing functional correctness. However, focusing solely on execution-based correctness can be misleading, as passing tests does not necessarily reflect the overall quality of the code [42]. Issues related to readability, maintainability, or performance may persist even when the code is functionally correct. To assess these non-functional code aspects, an emerging studies rely on static analysis tools to identify security, maintainability and readability issues and evaluate based on the occurrence of these issues [40, 41, 125]. Despite significant advancements in code generation benchmarks, the evolution of evaluation metrics has not kept pace. The existing metrics focus on code structure or functional correctness. Although some studies have explored some non-functional code aspects, a comprehensive evaluation framework spanning multiple code aspects is still lacking.

3.5.3 The need for LLMs Regression Testing

LLMs are evolving rapidly with emerging new capable models. However, more capable LLMs do not always guarantee improvements; they may introduce unexpected regressions [21], highlighting the need for systematic regression testing in LLMs. In traditional SE, regression testing ensures that changes—such as feature updates or bug fixes—do not introduce unintended behavior. This is typically achieved by re-executing unit tests [137]. However, unlike traditional SE, LLMs are non-deterministic, making it difficult to define and apply unit tests consistently [138]. To address regression risks in LLMs, RETAIN [139] was proposed as an interactive tool that supports LLM model update with regression testing. RETAIN includes two key components: (i) an interactive interface tailored for detecting regression testing and (ii) an error discovery module that provides textual descriptions of output differences between LLMs and suggests prompt modifications to reduce inconsistencies. While RETAIN is effective for general-purpose LLM regression testing, it focuses solely on textual differences, making it unsuitable for evaluating LLM-generated code. Code generation requires assessment beyond text comparison, considering structural, semantic, and executional properties that influence both functional correctness and non-functional aspects [44]. With the rapid evolution of LLMs, the growing risk of capability regression, and the increasing integration of LLM-generated code into production systems, regression testing has become an urgent need. However, this challenge remains unexplored in the current literature.

3.5.4 Identified Gaps and Proposed Direction

Although some studies have examined the quality of generated code either systematically or by proposing a taxonomy [127] of inefficiencies or root causes in buggy code. These works do not offer a comprehensive view of the broader range of inefficiencies that can occur in LLM-generated code. To address this gap, we propose a systematic categorization of inefficiencies that goes beyond correctness to include other key aspects of code quality, such as performance, readability, and maintainability 5. We observe that our proposed taxonomy intersects with prior taxonomies [127] primarily in two categories, along with their respective subcategories: *General Logic* and *Errors*, both of which contribute to buggy code. However, inefficiencies related to *Performance*, *Readability*, and *Maintainability* have received less attention, as they do not always result in bugs but can still degrade overall software quality. By addressing these overlooked inefficiencies, our taxonomy provides a more comprehensive inefficiency categorization, offering new insights into the challenges of integrating LLM-generated code into real-world development.

Although the risk of capability regression in LLMs is well recognized, and various benchmarks and metrics have emerged to evaluate their performance, most comparative studies in code generation still focus primarily on correctness scores [140, 141]. This narrow focus overlooks a wide range of inefficiencies—many of which are non-functional in nature, such as performance, readability, and maintainability—that are critical when LLM-generated code is used in production systems. To address this gap, we propose *ReCatcher* in Section 6.2, a code-specific regression testing framework designed to go beyond correctness. ReCatcher enables structured comparison of LLM-generated code across multiple quality dimensions and systematically detects regressions that extend beyond surface-level or textual differences.

3.6 Chapter Summary

In this chapter, we reviewed the literature on reliability assurance in MLOps. We began by discussing the limitations of drift detection in visual industrial inspection systems, highlighting existing gaps and motivating our proposed direction. We then examined challenges in LLMOps for code generation, reviewing relevant work, identifying open problems, and outlining the rationale behind our research contributions.

CHAPTER 4 TOWARDS RELIABLE CONTINUOUS TRAINING FOR DEEP LEARNING INSPECTION SYSTEMS

Industries are increasingly adopting ML and DL algorithms for automated visual inspection systems that can detect defects without relying on human inspectors or hand-crafted rule-based designs [142–145]. MLOps practices have emerged to manage these visual inspection systems to manage model life cycle [7]. Despite their promise, ensuring reliability in MLOps pipelines remains a significant challenge—especially in dynamic environments where input data can drift or change unpredictably. Statistically learned models, such as those used in DL-based visual inspection, may struggle to generalize when confronted with production inputs that differ from the in-distribution (ID) data seen during training [29, 146]. This mismatch can lead to silent performance degradation that goes undetected unless proactive maintenance strategies are in place. Production manufacturing environments are often dynamic and consequently prone to many variations, such as unexpected changes in luminosity, which add further complexity [147]. However collecting a comprehensive training dataset that covers all potential scenarios is not feasible, due to the inherent uncertainty of such variations and their uniqueness [148]. To mitigate these gaps, CT has become a widely adopted MLOps practice aiming to fine-tune models on recent production data to adapt them to changing conditions [149]. Reliable CT workflow requires that fresh training samples to follow the original data distribution and to be accompanied with ground truth labels. Although commonly practiced, using a model's predictions as self-generated labels can result in incorrect labels being incorporated to CT workflow, as even well-trained models may not perform perfectly when faced with novel inputs in real-world settings [150]. Another MLOps reliability issue stems from distribution drift, where novel inputs deviate from the ID. Drifted inputs may result in erroneous model behaviors, such as false overconfidence [151], and CT on drifted data—even with ground truth labels—does not guarantee that a retrained model will be reliable, since all hyperparameters and engineering choices were originally optimized for the ID setting. Therefore, CT should be conducted on novel datasets that are correctly labeled and belong to the ID distribution or closely similar distributions [152]. Failure to meet these CT prerequisites may compromise MLOps pipeline reliability and result in model performance degradation over time [153]. To address these challenges and reinforce reliability and trustworthiness within MLOps workflows, we develop a robust CT-based maintenance approach that updates DL models using reliable data selections through a two-stage filtering process. This aims to mitigate the risk of incorrectly labeled or drifted inputs being included in the dataset used for CT. Specifically, the confidence filter rejects all inputs with weak prediction confidence scores, as the model inherently discredits its self-generated labels for them. At this initial filtering stage, our objective is to eliminate corner cases that may exist on the boundaries of the ID and could still pose challenges for the model to generalize. The selected inputs with reliable confidence scores then pass through a follow-up drift filter that aims to filter out inputs that are substantially shifted from the ID data. Due to the complexity and high dimensionality of imaging data, capturing these adverse shifts involves a preliminary step of dimension reduction. To achieve this effectively, we combine two complementary embeddings to derive comprehensive and representative profiles for the images: (i) a Variational Auto-Encoder (VAE) to extract the most relevant semantic features from the images (i.e., retains what characterizes a given image from the other images within the data distribution), and (ii) a pixel histogram to generate a pixel profile as a numerical vector representing the image pixel-value characteristics, capturing tonal, color, and shade information. Combining these semantic and pixel-value profiles into an image embedding creates a new low-dimensional space. This space allows for computing distances between embeddings and efficiently learning statistical models. Typically, the ID data is available during model engineering, and data drift tends to occur after deployment in the production environment [154]. To simulate potential drifts, random data transformations with varying severity levels are commonly used. However, there is no guarantee that these transformed data are representative of all possible data drifts [155]. Therefore, we train a one-class classifier on the ID image embeddings to capture their patterns and distinguish them from out-of-distribution (OOD) image embeddings. The transformed data are used to validate that the optimized one-class classifier can indeed reject substantially shifted inputs based on their embeddings.

Subsequently, our proposed CT-based maintenance approach proceeds with a fine-tuning of the DL model on the filtered inputs while validating on a combination of recent production and original training datasets. This strategy mitigates catastrophic forgetting, i.e., degradation of performance on the original distribution data, and ensures the model adapts effectively to new operational conditions.

Our proposed two-stage filtering approach minimizes the risk of retraining the model on non-reliable data that are either incorrectly labeled or substantially shifted from the original training data distribution enhancing reliability MLOps pipeline reliability. We evaluate our approach on two real-world industrial use cases: the first examines potential defects in Popsicle stick prints, and the second investigates defects that may occur during the production of glass bottles. An assessment on critical real-world scenarios involving data drifts shows that our CT-based maintenance retains only 8% of incorrectly auto-labeled instances for fine-tuning the original model, and the updated model maintains its performance on production data but also enhances it by up to 14%, without compromising its initial results on the

original validation datasets.
The rest of this chapter is structured as follows: Section 4.1 delves into the details of our two-stage filtering approach for production data and CT workflow. Evaluation results are presented in Section 4.2. Section 4.3 presents potential threats to validity.

4.1 Proposed Approach

In this section, we describe the different steps required to co-design our proposed two-stage filtering approach for reliable self-improving visual inspection system.

4.1.1 Characterization of Risky Data

Supervised ML models are commonly estimated via empirical risk minimization (ERM) [156], a principle that considers minimizing the average loss on observed samples of data, as an empirical estimate of the true risk, i.e., the expected true loss for the entire input distribution. The average cost reduction leads to greedily absorbing the patterns that hold on the majority of training instances, resulting in a biased model that is prone to outliers and unfair to minorities [157]. Hence, the first category of risky data is the underrepresented inputs that are most likely to be erroneously classified by the model due to their infrequent occurrences in the ID data. Furthermore, ERM assumes that training and test data are identically and independently distributed (a.k.a. i.i.d. assumption) [158], which explains the use of held-out validation data as a proxy for unseen data points. However, data drifts often occur in production environments for many reasons, such as naturally-occurring shifts (unfamiliar background, motion blur, random noise, unexpected illumination, occlusion, etc.) [61,62,110], new emerging defects, old defect pattern variations, or deviations in defect statistics [59]. As the model may not generalize on drifted inputs, its overall predictive performance could suffer [77, 159]. Thus, the second category of risky data consists of OOD inputs, i.e., data that belong to the true distribution (i.e., operational domain of the inspection system), but they are absent from the training and validation datasets due to selection bias. Finally, the third category of risky inputs consists of anomalous data that may incidentally arise in real-world production situations, but they do not fall under the inspection system's foreseeable operating conditions [160], and therefore, should be identified and excluded.

4.1.2 Confidence-based filtering stage

In ML, confidence scores represent the model's estimated probability that a given input belongs to a particular data class. In other words, these scores represent how confident the model is in its predictions [161]. Higher confidence scores indicate a higher level of certainty in determining the class of a given input, whereas lower scores suggest uncertainty [56]. However, the inherent complexity of neural networks, coupled with the main objective of ERM being to maximize classification accuracy during training, contributes to the lack of calibration in model confidence outputs [162]. The calibration aligns the predicted confidence scores with the actual correctness rates, i.e., a model that is perfectly calibrated will predict a confidence score of 0.8 for samples with a correctness rate of 80% [163]. In particular, calibrating confidence scores is crucial to gauging the reliability of predictions and quantifying the model's uncertainty [164]. Confidence calibration techniques can be used at different stages: training, post-training, and inference [165]. Regularization, during-training calibration, is insufficient in dealing with overconfidence [162]. Ensemble approaches, during-inference calibration are computationally expensive and time-consuming, as they involve multiple models [166]. Post-training methods have been investigated with temperature scaling proving to be particularly simple and effective [162]. Indeed, a temperature parameter, $T > 0$ is used to re-scale logits before applying softmax, softmax $= \frac{\exp(z/T)}{\sum_i \exp(z_i/T)}$. The T is adjusted on the validation logits returned by the trained model by minimizing the Expected Calibration Error, which quantifies difference between prediction probabilities and real probabilities. In our approach, we used temperature scaling, which we found to be reliable and less prone to errors. It is also aligned with our continuous improvement philosophy, as its effectiveness can be enhanced with more validation data that helps in accurately estimating the temperature parameter and further improving the model's confidence scores. Overconfident situations, which are common in modern neural networks [167], where probabilities frequently skew to 0 or 1, can be mitigated by $T > 1$. Nonetheless, even calibrated confidence scores should be interpreted with caution, as they may fail to capture epistemic uncertainty in scenarios like data drift [168] and may suffer from reliability degradation. Therefore, we develop our first stage of data filtering based on confidence scores that targets the rejection of the underrepresented inputs. As this particular category of risky data belongs to ID but with less frequency, we believe that their corresponding calibrated scores would be relatively low due to the model's low accuracy on their input space regions [105]. Our confidence-based filter requires setting a single parameter—a threshold—to segregate low-confidence predictions from unseen inputs and prevent their inclusion in the self-labeled data. Tuning this threshold is crucial and involves a tradeoff: a higher threshold reduces false positives (incorrectly classified inputs), but increases false negatives (correctly classified inputs mistakenly rejected). Alternatively, a lower threshold maximizes true positives (correctly classified inputs), lowering false negatives but increasing false positives. Despite seeming counterintuitive, a lower threshold can improve overall results by allowing more inputs to proceed to our subsequent filtering stage, which can effectively remove the residual risky data while preserving enough correct samples for continuous training. Therefore, we propose to address the acceptable performance level of the inspection system as a primary consideration. Then, we select the lowest threshold where the achieved performance meets this criterion. This procedure enables us to tolerate false predictions in order to mitigate the risk of rejecting correct predictions, all while ensuring that we do not fall below our target performance level.

4.1.3 Distribution-based filtering stage

A major drawback of ML models, which is further accentuated by the rise of deep neural networks with superior learning capacity, lies in their propensity to always generate an output, as long as the inputs are provided in proper numerical form, even if such input is meaningless in the domain where the DNN is supposed to operate [29]. Likewise, we observe the inability of the models to distinguish the ID inputs from the ODD ones. This has led to the emergence of out of distribution/drift detection approaches that aim to complement the predictive models and overcome their limitations regarding self-recognition of their valid input domains. Following are the details of our proposed distribution-based filter that reveals OOD images in the context of visual inspection, i.e., the inputs do not fall within its operational domain.

Rich image embeddings

In the context of visual inspection system, high-resolution images inherently present a challenge due to their high-dimensional nature, making it difficult to measure similarity and dissimilarity between them [169]. As a solution, semantic feature extraction has gained traction in practice, which involves employing diverse methods, ranging from statistical techniques like Principal Component Analysis (PCA) to DL-based approaches like autoencoders [170]. The extraction of meaningful and condensed representations for the input images streamlines the data analysis by capturing essential information while reducing the dimensionality [171]. In our approach, we leverage VAE [172] to derive image embeddings that retain semantic variations across instances. As deep generative models, VAEs comprise two fundamental components: the encoder and the decoder. The encoder takes an input image and maps it to a latent representation distribution, then used by the decoder to reproduce the same image as output [172]. The learned latent representation encapsulates a valuable and condensed embedding for semantic image profiling because it encodes the images into a smooth, continuous, lower-dimensional space while preserving essential information to differentiate between them based on their semantic content [30]. Contrary to conventional autoencoders, VAEs employ stochastic encoding representation that captures the underlying distribution of the latent space [173], preserving the diverse aspects of the encoded images. Although this probabilistic aspect improves robustness, VAE models do have limitations. These models may suffer from posterior collapse, which occurs when the VAE encoder fails to generate accurate input embedding [174]. This leads to embedding overlap between ID and OOD inputs, and in more severe cases, the latent space can become independent of the input data. To address these limitations and enhance encoding capacity, we opted for fine-tuning VAE rather than training from scratch. This approach facilitates the learning of robust, high-level embedding representations. To create this pre-trained VAE, we opt for MVTec AD [81], which is a dataset for benchmarking industrial inspection models. It includes 5354 high-resolution images across fifteen object and texture categories, with each category containing defective and defect-free images. By fine-tuning the VAE on this rich defect dataset first, high-level domain patterns can be acquired, which are then transferred to the use-case-specific VAE, reducing the risk of posterior collapse and improving its OOD representation. As we aim as well, to distinguish the shifted data from ID data, VAE-generated image embeddings should be sensitive to semantic shifts. To accomplish this, we systematically simulate both moderate and significant shifts by applying diverse image transformations such as blurring, sharpening, and brightening at varying intensity levels (more details in Section 4.1.3). Subsequently, we measure the disparity between the embeddings of original images ("original embedding") and the embeddings of synthetically-shifted images ("shifted embedding") in order to assess their discriminative power. In fact, we calculate the average of the pairwise distances between all the original and shifted embeddings. We use the L_∞ as a distance measure that computes the maximum absolute difference, while averaging offers an overall assessment of the discrepancies between the two data distributions. As disparity levels increase, the derived image embeddings become more accurate at separating ID instances from their synthetically-shifted counterparts. While VAE embeddings are effective in capturing semantic differences among images, they may overlook fine-grained pixel value distributions [175], as it can manifest as random noise, color changes, and semantically-preserving variations. Nonetheless, such pixel-level variations can negatively affect the neural network behavior and leads to misclassification, as evidenced by the studies on adversarial attacks [176]. To consider pixel-value variations in our image embeddings, we developed an histogram profiler to capture the nuances of pixel value distributions. We start by calculating a condensed histogram that aggregates pixel occurrences within different buckets (ranges of values) for each color channel. Subsequently, the occurrences are normalized by dividing them by the total number of pixels (i.e., height $\times$ width), then, concatenated them sequentially in the order of the red, green, and blue channels. Therefore, our compact rich embedding is the concatenation of the VAE encoding profile and the histogram-based image profile.

Classification-based OOD detection

Once we construct the compact embedding space, the next step is to learn the patterns and characteristics of the ID embeddings compared to OOD embeddings. The use of supervised learning algorithm poses a significant challenge, arising from the necessity of explicitly identifying unknown potential future forms of distribution drift [177]. As an alternative, statistical testing can assess whether production data embeddings belong to the same distribution of the training data embeddings [106] [28]. Our derived embeddings serve as condensed representations of images, yet they may consist of hundreds of elements that do not conform to any specific underlying distribution. Thus, multivariate statistical tests, known for their sensitivity to slight changes [154], are rendered ineffective in our use case [178]. Another avenue within the realm of unsupervised anomaly detection involves the utilization of one-class classifiers (OCC), such as Isolation Forest (IF) or One-class Support Vector Machine (OC-SVM) [179]. These ML algorithms are trained on samples belonging to a single class. The resulting model is capable of delineating a boundary decision surrounding the training data distribution. At the inference, the model predicts whether a new input belongs to the learned data distribution or not [180]. These OCC models handle multi-dimensional data well, and they do not require OOD training samples. Typically, anomalous or drifted inputs are reserved for testing to evaluate the model performance. These one-class classifiers have shown their effectiveness at detecting anomalies and OOD instances [179, 181]. Our research focuses on identifying risky data to filter them out because the model may produce false (random) predictions, and even with correct predictions, their inclusion into training samples may adversely alter the model's inductive bias [152]. These risky data can manifest as unexpected or OOD inputs, as well as instances that are under-represented in the training data distribution. In conventional DL model engineering scenarios, all available data is typically considered during the design process. Consequently, it becomes challenging to anticipate drifted data, often requiring their collection post-deployment. To approximate shifted production data, we employ synthetic data transformations outlined below to generate shifted versions of the original samples. These transformations offer flexibility in introducing varying degrees of shift.

Blurring is achieved through the application of a Gaussian filter. The degree of blurring, reflecting the transformation's intensity, is adjusted by tuning the standard deviation (σ) within the Gaussian filter, thereby managing the spread of the underlying Gaussian function. A higher σ indicates a more aggressive transformation. A σ value of 0 renders the original image, and there is no upper limit for this value.

Brightening is achieved through the utilization of a Brightness adjustment filter, employing the brightness factor, denoted as β, as an intensity control. This ensures uniform brightness enhancements over the entire image. The higher β the more pronounced and aggressive the transformation becomes. Notably, β value of 0 results in a black image, a β of 1 renders the original image, and there is no upper limit for this value.

Sharpening is accomplished through the application of an Unsharp Masking transformation. This involves creating a degraded version of the image and then subtracting it from the original to obtain a sharpened image. The degree of sharpness is controlled by a sharpening factor, denoted as s , where a higher value corresponds to a more aggressive transformation. A s value of 0 produces a blurred image, while a factor of 1 maintains the original image.

Identifying the maximum level of distortion tolerated by a given model poses a challenge, as transformed inputs within a certain intensity range may either drift or remain acceptable based on the original data. To simplify the task, we opt for categorizing the synthetic data transformations into two distinct groups: moderate and harmful data shifts, relying on their intensities. Indeed, moderate shifts closely resemble the conventional data augmentation techniques [182] employed to enrich training datasets with inputs derived within the in-distribution, thereby increasing model generalizability. Conversely, harmful shifts produce synthetic OOD data that are beyond the model's limits, but may result from unfavorable production conditions. They are prone to inducing erroneous predictions and instabilities during retraining, ultimately resulting in deviations from the model's intended behavior. To determine the intensity level of each shift, we initiate by employing default intensities commonly used in data augmentations [182], then refining them through qualitative validation (i.e., human-in-the-loop). Indeed, we manually examine the input semantics before and after each transformation w.r.t the data distribution in order to ensure the preservation of relevant information necessary for the prediction task. This human-in-the-loop validation process draws upon existing research on semantically-preserving transformations [183, 184]. To construct the distribution-based filter, we train an IF model on our compact embeddings of original training samples. Then, we assess the trained IF model on the produced synthetic dataset. The assessment reposes on estimating the acceptance and rejection rates for both categories: moderately-shifted and harmfully-shifted inputs. The optimal IF model is one that can simultaneously maximize the rejection rate for harmful shifts and the acceptance rate for moderate shifts. Hence, our distribution-based filter can effectively reject all drifted data (represented by harmful shifts), wherein the inspection system is prone to erroneous predictions, while still accepting incoming inputs closely resembling the in-distribution data (represented by moderate shifts).

4.1.4 Continuous Training-driven Maintenance

In this present research, we implement a reliable CT-driven maintenance approach for DL-based inspection systems. For such systems, the vanilla CT with self-generated labels can be effective when production data are aligned with the training samples' distribution [152]. However, real-world industrial environments introduce dynamic variations such as changes in luminosity, vibration patterns affecting object positions, and machinery depreciation-induced behavioral shifts [147]. In response to these environmental variations, we expect DL-based inspection systems to maintain a certain performance level due to their inherent robustness to data distortions and noise that do not significantly compromise semantic information. However, vanilla CT with self-generated labels fails to enhance and even maintain inspection model performance amidst such environmental fluctuations, as the number of false predictions increases and incoming inputs may substantially deviate from the in-distribution samples [152]. Therefore, a preliminary filtering step is crucial to exclude risky data points (i.e., false predictions or drifted inputs) from the auto-labeled datasets supplied to CT. Initiating CT can be prompted by meeting a specified condition, such as after a pre-defined time period, performance degradation triggers, on-demand retraining, or a combination of these conditions [50]. In our context of supervised industrial inspection, the monitoring of performance degradation depends on groundtruth labels and the on-demand retraining requires ML expert oversight. Hence, periodic CT can be streamlined by synchronizing the intervals with production pauses and scheduling maintenance within a time window that allows for the collection of an adequate number of production entries. In between CT cycles, the DL-based inspection system logs all inputs alongside their model-generated predictions, i.e., labels and calibrated confidence scores. Upon CT triggering, a confidence-based filter is employed to discard inputs associated with low (calibrated) confidence scores, specifically those below a predefined threshold carefully tuned on testing datasets. Subsequently, the remaining inputs with adequate confidence levels undergo a distribution-based filter. This filter generates rich embeddings of the inputs, which are then fed into an Isolation Forest (IF) model optimized to reject any embedding substantially shifted from the in-distribution embeddings. As a result, the final retained inputs are more likely to be correctly classified and sufficiently similar to the original samples (i.e, no significant deviation from the in-distribution). Thanks to transfer learning, the CT-driven maintenance of DL-based inspection system does not require the retraining of the model from scratch on the entire dataset, i.e., combining the original and production entries. Instead, it operates directly on the last production model, considering it as the base (pretrained) model, and fine-tunes it exclusively on the filtered production data using the same hyperparameters, except for decreasing the maximum number of epochs. This reduction helps mitigating the risk of catastrophic forgetting phenomenon, where the model overfits to the new samples to the extent of losing its original performance. Furthermore, a merged validation dataset that includes both of original and novel validation data (gathered after deployment) also contributes to prevent catastrophic forgetting. Indeed, the resulting comprehensive assessment ensures that the model adequately handles both new and past entries, contributing to sustained performance across CT cycles. It is worth noting that the new validation dataset was derived by partitioning the production data according to the same training/validation split ratios utilized in the initial training process. This approach ensures a proportional representation of the current and historical sources of training data for continuous training (CT).

4.2 Evaluation

In this section, we detail our evaluation setup including use cases, models, metrics, and procedures. Next, we present the evaluation results through examining four research questions.

4.2.1 Experimental Setup

Use Cases

The datasets for our use cases consist of proprietary images from industrial partners, which we cannot share due to non-disclosure agreements. Instead, we describe the use cases, datasets, and data collection and preparation strategies.

Popsicle Stick Prints (POP). The task involves the quality inspection of logos and text prints on Popsicle wood sticks based on a single top camera view. The inspection focus on the precision and clarity in the logo and the overall readability of text, e.g., a pale-colored logos or text prints should be considered as a defect in printing. An illustration of Popsicle stick is presented in Figure 4.1.

POP Original Dataset. It includes 1212 images, of which 700 represent good stick prints, and 512 exhibited defective stick prints. These images, captured at a resolution of 500x115 pixels, were obtained during the development phase and were divided into 80% training and 20% validation samples.

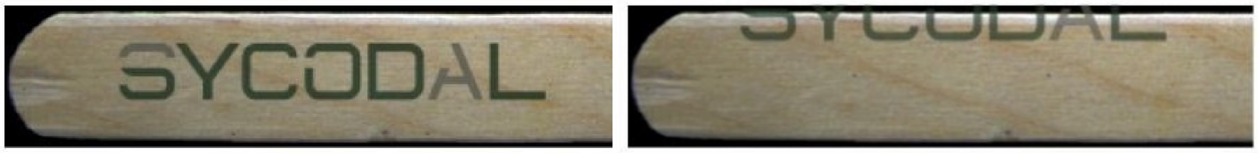


Figure 4.1 Good vs Decentralized Popsicle Stick Print

POP Production Dataset. It represents a dataset collected in production environment (on-site testing phase) within a critical window time, during which the model experiences a severe degradation in its predictive performance. This critical production dataset contains 356 images.

Glass Bottles (GB). This task is concerned with the quality inspection in glass bottles filled in packaging boxes based on a single top camera view. Depending on the packaging strategy, bottles can be filled in two orientations: either upright, revealing the bottleneck, or inverted, showcasing the bottom of the glass bottle. Whenever one bottle is the inverse of what is expected, it is regarded as a defect. For bottle defects, we found the occurrence of broken bottle finish when the bottle exposes its neck on the top, and the occurrence of cracked bottom when the bottle is turned upside down. An illustration of these defects is presented in 4.2. *GB Original Dataset.* It comprises 1195 images, wherein 597 depicted good bottles without any defects, while 598 showcased bottles with defects either at the bottom or the finish of the neck. The resolution of the images is on average of 224*224 pixels. Indeed, these images of good/defected bottles are collected by cropping an original image of 6-glass bottle packaging box, captured at a resolution of 1024*1024, into 6 sub-images containing each a bottle.

GB Production Dataset. Similar to the POP use case, we collected at on-site testing phase, a subset of critical dataset on which the model poorly behaves. We obtained a total of 455 images gathered within a specific window time.

Image Transformation Settings

We set the following factors after validating by experts for the image transformations described in 4.1.3:

Moderate Shift. β: (1.5 and 1.8), s: (5 and 20) for both use cases, and for σ :(2 and 3) and (0.8 and 1) for POP and GB use case respectively.

Harmful Shift. β: (2.5 and 3), s: (100 and 500) and σ: (5 and 10) and for both use cases.

We apply these transformation to generate two datasets: moderate and harmful synthetic samples for VAE pre-training and OCC validation.

Models

Binary Classification. For both use cases, the inspection system relies on a binary classifier leveraging ResNet [185] pretrained model sizes: 18, 50 and 101. Best results are given by ResNet18. The optimal hyperparameters are the following: learning rate of 10^{-4} and a batch size of 32, spanning 200 epochs with early-stopping, Adam as optimization algorithm and

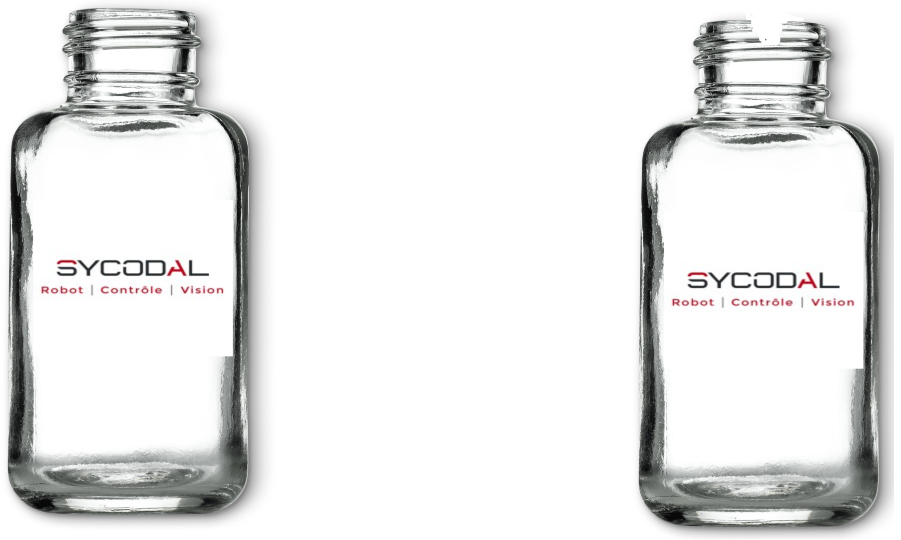


Figure 4.2 Defective vs Non-Defective Glass Bottle

Cross Entropy as loss function. For fine-tuning during CT, we reduce the epochs to 50 to avoid catastrophic forgetting as only the entries from the last time window are used.
VAE. It comprises an encoder with three convolutional layers succeeded by max-pooling layers, followed by a flatten and two hidden layers and a decoder mirroring the reverse structure of the encoder. All layers employ the Rectified Linear Unit (ReLU) as neuron activation. We use Adam as optimizer and Mean Squared Error as loss function, as well as a learning rate of 10^{-4}, a batch size of 32, and spanning 200 epochs. We explore various latent space dimensions, namely 64, 128, 256, 512, and 1024.
OCC. We select Isolation Forest (IF) with tuning hyperparameters using: (i) proportion samples used range from 0.1 to 0.9 with a step of 0.1; (ii) contamination proportion range from 0.01 to 0.4 with a step of 0.05, maximum number of features range from 0.1 to 1.0 with a step of 0.1.

Evaluation Procedure

We conduct multiple experiments, repeated at least 5 times to secure robust set of results. Below are the the metrics and the procedure used.
F1-score (F1) is the harmonic mean of precision and recall to balance between these two metrics.
%True Acceptance (TA) denotes the ratio of correctly-classified instances that successfully pass a given filter.
%False Acceptance (FA) represents the ratio of incorrectly-classified instances that successfully pass a given filter.

%True Rejection (TR) indicates the ratio of incorrectly-classified instances that are both rejected a given filter.
%False Rejection (FR) indicates the ratio of correctly-classified instances that are rejected by a given filter.
Dispersion. consists of the average of the maximum absolute differences between two vector distribution, which can be formulated as follows: $\|U - V\| = avg(max_u(\sum_v \|u - v\|_\infty))$.
Qualitative. Data were sampled randomly from the various groups based on their label correctness and associated two-stage filter decisions: pass or reject. Then, we provide them to two domain experts for qualitative analyses of the possible causes of false predictions or rejections by filter. For both studied use cases, a summary of the analysis' outcomes is added to help understand the findings.

4.2.2 Production Environment

The engineering team targets a performance of 95% in terms of F1-score, but requires a minimum of 90% of validation F1-score to deploy the model for testing in the production. However, following the deployment, the system experienced performance decrease, during which we collected production data. Table 4.1 summarizes model performance during the development process and on (critical) production data.

4.2.3 Experimental Results

To evaluate our proposed approach, we studied the following research questions:

RQ.1: How effectively can confidence-based filtering be used to reject false predictions?

Motivation. Calibrated confidence scores can indicate how certain the model is in its predictions. We aim to assess their effectiveness in filtering out the inputs associated with unreliable model's outcomes.
Method. Given the established performance of 95% F1-score, our procedure (Section 4.1.2) is to select the lowest threshold for confidence scores, where the model achieved the target

Table 4.1 Binary Classifier Model Performance

Use case	Validation F1	Production F1
POP	93.2%	45.7%
GB	93.3%	76.6%

score on the validation dataset. Then, we compute the true and false acceptance rate (TA and FA) as well as the true and false rejection rate (TR and FR) obtained by the designed filters on production data collected for both use cases. The assessment of the confidence-based filter effectiveness is conducted on the (critical) production dataset. This shows their capabilities as they were deployed in production when these critical inputs occurred.

Results. Table 4.2 shows, on one hand, the results of confidence-based filter on validation datasets after selecting the most adequate threshold. On the other hand, it reports the same metrics on production datasets that are degraded, i.e., the true acceptance rate dropped while the false acceptance increased by 5 times for POP and 10 times for GB. Nevertheless, the rejection rates are relatively low and stable among the experiments. This reinforces the overconfidence phenomenon where the model tends to assign high confidence scores even for unusual, novel data. A follow-up filtering stage is needed to further remove the passed inputs labeled by false predictions in order to prepare for a reliable CT.

Finding: Confidence-based filters tend to accept high proportion of inputs at production, despite the increased risk of false acceptance.

RQ.2: Can the proposed embeddings serve as features to separate between shifted and original data inputs?

Motivation. The aim is to ensure that the designed embeddings can distinguish the incoming inputs based on their degree of deviation from the in-distribution data, i.e., original samples.

Method. First, we generate input samples form original dataset, moderately-shifted dataset, harmfully-shifted dataset. Then, we compute their respective embeddings using the proposed encoding techniques: VAE and Histogram. We vary the embedding space dimensions for VAE and Histogram by changing the latent space size and the pixel bucket size, respectively. Once the embeddings are computed, we estimate the dispersion between the original embeddings and the moderately-shifted embeddings, denoted O.-vs-M., and the dispersion between the original embeddings and the harmfully-shifted embeddings, denoted O.-vs-H., using the in-

Table 4.2 Confidence Filter Performance

Use case	Thresh.	Dataset	TA	FA	TR	FR
POP	0.79	Validation	87.5%	3.3%	2.1%	7.1%
		Production	65.7%	**16.6%**	7.1%	10.6%
GB	0.58	Validation	91.9%	3.8%	2.6%	1.7%
		Production	65.5%	**30.5%**	2.2%	1.8%

troduced measure. The higher the dispersion between the originals and shifted ones, the better the embedding method will be at capturing discriminative information.
Results. Table 4.3 reports the dispersion measurements for both VAE experiments: O.-vs-M. and O.-vs-H., with increasing latent space dimension, namely, 1024, 512, 256, 128, and 64. In similar manner, Table 4.4 shows the results of Histogram experiments with increasing bucket size, explicitly, 10, 20, and 30. As expected, both encoding techniques show a dispersion between the original and the moderately-shifted embeddings higher than the dispersion between the original and the harmfully-shifted embeddings. As a result, this demonstrates their ability to retain relevant information in order to capture potential distribution shifts. While all the proposed encodings can differ originals from the shifted versions, we found that the increasing of the space size, i.e., latent features or bucket size, causes the reduction of dispersion measures, which reflects a negative impact on the discriminative power of the created embeddings for both use cases. Therefore, we decided to proceed with the reduced dimensional size of both VAE and Histogram, 64 latent features and 10 bucket size, that showed the maximum dispersion between the two use cases. In addition, we will merge them into a concatenated feature vector to enrich the information for the one-class classification.
Finding: The proposed VAE and Histogram encoding methods produce dimensionally-reduced image embeddings, capturing distribution shifts effectively.

RQ.3: To which extent can the proposed distribution-based filter complement the confidence-based filter for more reliable input selection?

Motivation. Even after calibration, confidence-based filters do not perform as well on validation datasets as they do on production datasets. By examining a follow-up filtering stage based on distribution, we aim to determine whether data drift is responsible for this reliability degradation.

Table 4.3 VAE Encoding Dispersion

Latent Dimension	Use case			
	POP		GB	
	O.-vs-M	O.-vs-H	O.-vs-M	O.-vs-H
1024	1.45	1.73	2.43	2.82
512	1.38	1.67	2.13	2.43
256	1.44	1.85	2.74	3.08
128	1.56	1.81	2.8	2.98
64	**1.64**	**1.95**	**4.013**	**4.49**

Table 4.4 Histogram Encoding Dispersion

Bucket Size	Use case			
	POP		GB	
	O.-vs-M	O.-vs-H	O.-vs-M	O.-vs-H
10	**0.51**	**0.82**	**0.33**	**0.56**
20	0.36	0.55	0.26	0.38
30	0.28	0.43	0.21	0.29

Method. We follow the same method performed for the confidence-based filter evaluation. We assess the performance of the filter on the production dataset in terms of true and false acceptance rate (TA and FA), as well as true and false rejection rates (TR and FR). As both encoding techniques have shown promising results, we train the IF model on different image embeddings: VAE, Histogram, and VAE + Histogram (a vector concatenation), and we compare their respective performances as input features for the distribution-based filter. Training the IF model is carried out using the original samples but the validation is conducted on the transformed inputs, simulating moderate and harmful data shifts. After the assessment of distribution-based filter (alone) on the production dataset, we assess the performance of the end-to-end filter, sequentially applying the confidence-based filter first on the production dataset and then the distribution-based filter on solely the passed inputs. We can therefore study the complementarity of both filters.

Results. Table 4.5 reports the performance results of distribution-based filter on the entire production data using different image embeddings for both use cases. As expected, the concatenation of the embeddings from the proposed two encoding techniques (VAE + Histogram) yields the best performance rates, as the one-class classifier has access to more discriminative and diverse information about the images to capture the in-distribution data characteristics. The VAE or Histogram encoding yields lower true and false acceptance rates when used alone for the POP use case. For the GB use case, they result in higher false acceptance rates and lower true acceptance rates. The concatenation of their embeddings, however, shows their complementarity and provides the highest true acceptance rates with relatively low false acceptance rates. If we compare the results of distribution-based filter in Table 4.5 with the results of confidence-based filter in Table 4.2, we found that the distribution-based filter tends to reject higher number of inputs than the confidence-based filter. As a result, both true and false acceptance rates are modestly reduced for POP use case, indicating that the data shifts are not pronounced in production data. In contrast, both true and false acceptance rates are substantially decreased for GB use case, suggesting that production data are indeed shifting. It is worth noting that despite the data shifts, true and false rejection rates are comparable, respectively, 25% and 34%, for GB use case. This demonstrates that the distribution shift does not induce systematically a false prediction, and the model tends to produce as many correct predictions as incorrect ones for inputs rejected based on their deviation from original data. Nevertheless, correctly-labeled, drifted data are considered unreliable for continuous training because they alter model patterns and cause catastrophic forgetting.

Finding: The VAE and Histogram methods can be combined to produce a rich, merged embedding that can serve as feature to learn a OCC effective in distinguishing between ID and OOD instances.

Table 4.6 reports the performance results of the proposed two-stage filter, which combines both confidence-based and distribution-based filters, in order to pass only the inputs with confident predictions and likely belonging to the in-distribution. For both use cases, the true and false acceptance rates (TA, FA) are further reduced, resulting in (50%, 8.85%) and (31.42%, 5.49%) for, respectively, POP and GB. This means that the false labels, 8.85% and 5.49%, represent almost 15% of the entire passed proportion data, 58.85% and 36.91% for, respectively, POP and GB. However, the two-stage filter manages to pass over half of the production data for the POP use case, but almost a third for the GB use case. The proportion of passed data is relevant because the data size has effect on the effectiveness of the subsequent model training iterations. In fact, the confidence-based filter passes more production data for GB than for POP use case, but the false acceptance rate was the half of the true acceptance rate, which shows a pronounced overconfidence issue (see Table 4.2).The distribution-based filter alone leads to a low acceptance rate in the GB use case (see Table 4.5). The two-stage filter is affected directly by this reduction in acceptance rate (see Table 4.6). We can conclude that the critical production data collected for GB use case was severely drifted.

Finding: The distribution-based filter successfully complements the confidence-based filter to lower false acceptances by up to six times, since it targets all accepted inputs with overconfident scores.

Table 4.5 Distribution-based Filter Results

Use case	Encoding	TA	FA	TR	FR
POP	VAE Enc	18.6%	3.4%	20.3%	57.7%
	(VAE + Hist) Enc	57.7%	13.4%	10.3%	18.6%
	Hist Enc	25.4%	8.0%	15.7%	50.9%
GB	VAE Enc	32.7%	14.1%	18.7%	34.5%
	(VAE + Hist) Enc	33.2%	7.7%	25.1%	34.1%
	Hist Enc	13.0%	16.5%	16.3%	54.3%

Table 4.6 Two-stage Filter Results

Use case	Encoding	TA	FA	TR	FR
POP	VAE Enc	16.3%	2.0%	21.7%	60.0%
	(VAE + Hist) Enc	50.3%	8.9%	14.9%	26.0%
	Hist Enc	21.4%	5.4%	18.3%	54.9%
GB	VAE Enc	31.2%	12.1%	20.7%	36.0%
	(VAE + Hist) Enc	31.4%	5.5%	27.3%	35.9%
	Hist Enc	11.9%	14.5%	18.2%	55.4%

RQ.4: Can CT on filtered data outperform vanilla CT in terms of F1-score enhancement after degradation?

Motivation. The main objective of the two-stage filter is to retain a valid subset of inputs that can be used to do CT-driven maintenance of the inspection system after performance degradation. Thus, we compare the performance improvement of the fine-tuned model with our proposed reliable CT to conventional (vanilla) CT.

Method. To perform CT, we propose to fine-tune the model in production using the novel data with self-generated labels, which can be divided into training (80%) and validation (20%) datasets. The difference between our proposed reliable CT and vanilla CT is the definition of novel data. A vanilla CT considers the production dataset collected within the last time window, i.e., since the last CT cycle. The reliable CT we propose uses a two-stage filtering strategy to remove risky instances from the production data, and feeds only a subset of the data into the model for fine-tuning. To compare the effectiveness of both CT approaches, we assess the performance of the fine-tuned in terms of F1-score based on two datasets: the first is the original validation set, which assesses the degree of induced catastrophic forgetting, and the second is the entire production set, which assesses the fine-tuned model's ability to cope with the latest conditions in the production environment that may remain for the foreseeable future.

Results. Table 4.7 shows the results of validation F1 and production F1, which represent the F1 obtained by the fine-tuned model on the original validation dataset and the entire production dataset, respectively. The F1 scores are reported by CT approach, either vanilla or our proposal, and by industrial inspection use case. Table 4.7 also includes the initial performance obtained by the original model as baseline for each use case. For POP use case, the original model has experienced a drastic decline in its F1 in production. According to CT results, vanilla approach slightly increased production F1 while almost decreasing validation F1 by the same percentage. Our proposed approach was able to increase production

F1 more than it decreased validation F1. The analyses we made on our filtering results led us to conclude that the confidence scores have been diminished for POP, indicating that the model faces challenging inputs because most of the risky data has been discarded at the first filtering stage. Based on this, it is likely that the original model overfits the initial data and that its learned patterns won't hold up against the incoming inputs. Unfiltered self-generated labels may include a high number of false predictions, so Vanilla CT cannot fix the overfitting issue. In contrast, our reliable approach reduces the overfitting problem by selecting half of the production data with correct labels and high confidence to reinforce the model patterns at the deployment environment. In the GB case, vanilla CT not only failed to improve the fine-tuned model's performance but also caused a catastrophic forgetting with more than 13% decay in the validation F1 data. In contrast, our proposed reliable CT succeeded to enhance both F1-score for original validation and production datasets. As we discussed during our analysis of filtering strategy assessment, the production data for GB shows severe drifts, leading to almost two-thirds of production inputs being rejected. Consequently, vanilla CT absorbs all production data associated with self-generated labels, leading to performance degradation on both original and novel distributions. As an alternative, our proposed approach relies only on a subset of data that remain close to the distribution and is confident that the self-generated labels will hold up, making the fine-tuned model more resilient against forgetting the original inductive bias and more capable of capturing the patterns required to cope with potential drifted inputs.

Expert Feedback.

POP Use Case Discussion: Domain experts analyzed the images that the model could not handle. Based on the defects collected during the model development, pale-colored prints, decentralised texts, and erased logo parts are the most common issues. It is all related to incomplete or missing parts of the object of interest, which is the wood stick's logo or text. Model degradation occurred in production when the ink dispatcher of the printer machine was slightly damaged, resulting in smudged prints. This results in a new challenging type of defect (illustrated by Figure 4.3) that is correlated with wrongly highlighted text or logo in a non appropriate way. This uncovered type of defect causes the model to behave poorly. Most of the images that were classified correctly have a mix of issues, as the prints were decentralised or there was uneven ink distribution between letters, which the original model can spot as defected. Despite correctly classified data being available, the volume was not sufficient to achieve acceptable performance through CT. Nevertheless, The fine-tuned model outperforms the original model on recent production data at the cost of performance degradation on the original validation data. Deploying the fine-tuned model is still advisable, as future data will likely match the recent distribution, ensuring the model adapts to current operational conditions. Meanwhile, alerting the maintenance team about the degradation allows them to investigate and address the root cause, maintaining overall model reliability.

GB Use Case Discussion: Domain experts noticed two changes in the production environment. First, ambient lighting on-premises led to data shifts due to inadequate camera settings in terms of exposure. Second, a new defect was introduced in the production line, which resulted in a dramatic drift in input data. It is characterized by an open mark in the bottle's body, as shown in Figure 4.4. The model predictions on these images are almost random due to the shifted location and form of defects, resulting in almost equal chances of correct and incorrect classifications. Thus, vanilla CT degrades both validation and production scores due to the prevalence of corrupted labels. The proposed approach, however, manages to discard the majority of these new defected products and fine-tune the model based on images that are shifted in brightness, resulting in an increased model robustness. In this use case, it outperforms the vanilla CT with a modest improvement. Importantly, by trimming the risky data, we prevent the catastrophic forgetting that causes substantial degradation in the fine-tuned model's performance on the original validation data when using vanilla CT. This careful data filtering strategy ensures more stable and reliable performance over time.

Finding: Our proposed CT-driven maintenance approach strengthens the positive impact of vanilla CT, while reducing the risk of performance degradation caused by high false prediction rates.

4.3 Threats to Validity

In this paper, we proposed an approach to enhance the automated selection of relevant production data for CT purposes. Our approach demonstrated its efficiency across two use cases in critical scenarios. Despite the promising results, several threats to validity exist.

External Threats. Our experiments are limited to visual inspection, which limits the generalizability of results. To address this, we opted for two different use cases POP and GB that are different in terms of task complexity, product, texture, dimension and defect types and we repeated each execution at least 5 times.

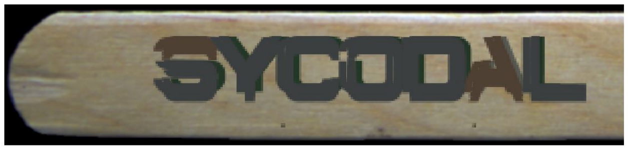


Figure 4.3 Smudged Stick Text Print

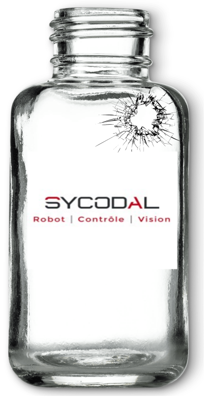


Figure 4.4 Glass Bottle: Open Mark Defect

Table 4.7 Continuous Training Results

Use case		Validation F1	Production F1
POP	Original Model	93.2%	45.8%
	Vanilla CT	91.7%	48.8%
	Our Approach	88.0%	59.7%
GB	Original Model	93.3%	76.6%
	Vanilla CT	80.2%	71.4%
	Our Approach	93.8%	79.1%

Internal threats. Confidence filter design involves the use of softmax despite the existence of multiple confidence scores such as entropy, perplexity score etc. This choice is being motivated by the wide use and proven performance of softmax in classification tasks [186], and to address its limits regarding overconfidence, we opted for calibration. Since the absence of drifted data by design and to enhance models generalization we opted for image transformation augmentation known for being unrealistic [187], to address this we relied on domain experts to define transformations and respective parameters. Finally, although our approach requires domain-expert involvement during approach setup, this involvement is not frequently needed and it ensures setup reliability.

4.4 Chapter Summary

This chapter addressed a key challenge in ensuring reliability within MLOps pipelines: maintaining deep learning-based visual inspection systems in dynamic and unpredictable production environments. We proposed a novel two-stage data filtering strategy to mitigate the risks associated with CT on auto-labeled fresh production data. First, we use calibrated confidence scores to eliminate risky data with high uncertain predictions. Second, we employ VAE and Histogram to encode rich image embeddings serving for OOD model optimization to filter significantly shifted risky data. The accepted subset is suitable for reliable maintenance of inspection system models. Our evaluation on two industrial inspection systems demonstrates that our approach passes less than 9% of false predictions, leading to an increase of F1-score up to 14% on critical production data. We discover limitations of supervised learning algorithms such as their inability to self-learn new defects and their reliance on human verification. Even if our reliable CT approach is applied for successive cycles, the model could not capture new defect patterns that are not similar to the ones from in-distribution.

CHAPTER 5 TOWARDS RELIABLE LLMOPS PIPELINE FOR CODE GENERATION: A TAXONOMY OF INEFFICIENCIES

While LLMs demonstrated promising results in code generation across various programming languages [37, 188–190], their generated code, similar to human-written code, exhibit various quality issues [191, 192], raising concerns about the reliability and trustworthiness of LLMOps pipelines, which are expected to integrate and deliver high-quality model outputs in production environments.

For instance, LLM-generated code has shown to suffer from redundancies, unnecessary computations, and suboptimal implementations [42], resulting in increased execution time, higher memory consumption, maintainability challenges, and import errors [43,193]. Previous studies have also shown that LLMs may generate incorrect logic, misinterpret task requirements, and struggle to handle corner cases, leading to critical issues [127]. Even when functionally correct, LLM-generated code can face quality issues [123]. It may be difficult to maintain [43], unnecessarily complex [42], prone to security vulnerabilities, and require manual intervention before deployment. These issues pose significant risks to software quality, particularly for developers adopting LLMs in real-world settings [194].

While prior research has examined specific quality aspects, such as correctness, security, and maintainability in LLM-generated code [40, 41, 123], inefficiencies as a broader concept remain largely unexplored. Existing studies focus only on buggy-generated code [123, 127] to categorize bugs and identify root causes. However, this narrow focus overlooks a critical issue: *even functionally correct code can exhibit inefficiencies that degrade software quality, such as performance bottlenecks [42], limiting real-world adoption of LLM-generated code.*

In this study, we aim to bridge this gap by identifying, categorizing, and analyzing overall inefficiency patterns, regardless of correctness, in LLM-generated code from three leading open-source LLMs for code generation. In this work, we define inefficiencies as any issues in LLM-generated code that degrade its quality, hinder integration into larger systems, or even affect functionality in isolation. Understanding inefficiencies guides the development of more efficient LLMs and helps practitioners and researchers improve LLM-generated code. Our investigation is guided by the following Research Questions (RQs):

- ***RQ1*** *What inefficiency patterns occur in LLM-generated code?*
- ***RQ2*** *How relevant are these inefficiencies to software practitioners and researchers?*

We conduct our study using *HumanEval+* a widely used benchmark for code generation composed of 164 human-crafted Python tasks [37]. We selected three leading open-source LLM models -CodeLlama, DeepSeek-Coder, and CodeGemma- to generate code for the different tasks, resulting in 492 generated code snippets. To evaluate the quality and requirements conformity of the LLM-generated code, we employed the higher-capacity GPT-4o-mini model as a judge. The judgments provided initial guidelines for our manual exploratory and qualitative analysis to identify inefficiency patterns in LLM-generated code. We adopted open coding [195] to iteratively derive categories of inefficiencies. Our study results in a taxonomy spanning five categories: *General Logic*, *Performance*, *Readability*, *Maintainability*, and *Errors*, comprising 19 subcategories. We validate this taxonomy through a survey with 58 software practitioners and researchers who use LLMs for coding assistance.

Our findings indicate that 33.54% of the studied sample exhibited multiple inefficiencies, indicating that inefficiencies in LLM-generated code are diverse and interconnected. *General Logic* and *Performance* inefficiencies are the most frequent and often co-occurring with *Maintainability*, and *Readability* inefficiencies. This co-occurrence suggests that limitations in LLMs' ability to generate correct and efficient code contribute to broader inefficiencies affecting code quality. The surveyed participants largely validated the relevance and prevalence of identified inefficiencies, confirming that it is aligned with real-world inefficiencies in LLM-generated code regardless of the model used. Additionally, while participants acknowledged these inefficiencies, they emphasized that in practice, they prioritize correctness over other code aspects. Our findings highlight a critical gap in LLMs' capability to generate correct, optimized, and high-quality code.

These results expose a critical limitation in current LLMOps workflows: a lack of mechanisms to robustly evaluate and assure pipeline-level reliability across LLM model updates, especially in terms of non-functional quality dimensions like efficiency.

By offering a structured taxonomy of inefficiencies and validating it through practitioner input, this study provides the necessary foundation for systematic quality assessment of LLM-generated code. This, in turn, enables LLMOps pipelines to incorporate more targeted, comprehensive evaluation strategies—making them more robust and reliable. It supports the development of quality gates, automated inefficiency detection, and more informed model versioning decisions, thereby directly enhancing the operational reliability of LLMOps pipelines.

To summarize, in this chapter we make the following contributions:

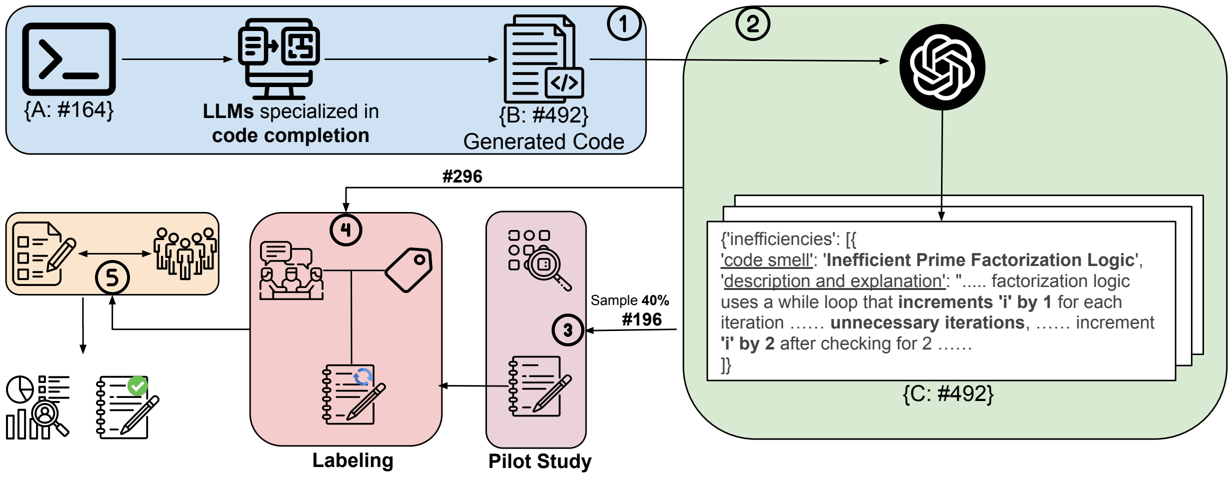
- We study inefficiencies observed in LLM-generated code and their prevalence.
- We propose a taxonomy of inefficiencies in LLM-generated code based on their characteristics.
- We validate our findings using an online survey and provide insights for researchers and

practitioners.

- We make our data and results publicly available in our replication package [196].

The rest of this section is organized as follows: In Section 5.1 we detail the methodology used to construct and validate the taxonomy of inefficiencies. In Section 5.2, we report our empirical findings, presenting our taxonomy. Then, we discuss threats to the validity of our results in Section 5.3 we we conclude the study takeaways in 5.4.

5.1 Methodology



1
{A: #164}
LLMs specialized in code completion
{B: #492}
Generated Code
2
#296
4
5
Labeling
Sample 40%
#196
3
Pilot Study
{'inefficiencies': [{
'code smell': 'Inefficient Prime Factorization Logic',
'description and explanation': "..... factorization logic uses a while loop that increments 'i' by 1 for each iteration ...... unnecessary iterations, ... increment 'i' by 2 after checking for 2 ......
}]}
{C: #492}

Figure 5.1 The methodology we followed for our study.

This section presents the methodology used in this study to construct and validate the taxonomy of inefficiencies in LLM-generated code. The process consists of five steps, as illustrated in Figure 5.1. First, we generated code using the HumanEval+ benchmark [37] and collected data from three open-source LLM4Code models: CodeLlama [89], DeepSeek-Coder [90], and CodeGemma [92]. Second, we leveraged a higher-capacity model (GPT-4o-mini) to judge the generated code and identify existing inefficiencies. In the third and fourth steps, we manually analyzed the code samples and the model's judgments to develop a comprehensive taxonomy. Finally, in the fifth step, we conducted a survey with practitioners and researchers to validate the LLM-generated code inefficiencies identified in our taxonomy. The details of each step are presented in the remainder of this section.

5.1.1 Step 1: Data Selection and Collection

Regarding the data collection, considering the popularity and wide adoption of Python as a programming language [197], this study focused on Python-generated LLM code. Given the focus on Python, we selected HumanEval+ [37] (denoted as A in Figure 5.1), a widely recognized and extensively employed benchmark for code generation [23, 198–200]. HumanEval+ enhances the original HumanEval [38] dataset by augmenting it with additional test cases, enabling a more rigorous and reliable assessment of the functional correctness of the generated code [37]. HumanEval+ comprises 146 Python programming tasks, each accompanied by (i) a task description provided as a doc string and (ii) associated unit tests to rigorously evaluate the correctness of the code snippets. Regarding the LLMs, we aimed to select those trained or fine-tuned for code generation tasks to enable a more meaningful and targeted analysis of inefficiencies. To ensure accessibility and feasibility, we established two selection criteria: (i) the models must be open source and (ii) have fewer than 8B parameters. Based on these criteria, we selected the three highest-performing models according to the HumanEval+ leaderboard [93]: CodeLlama-7B [201], GodeGemma-7B [202], and DeepSeek Coder-6.7B [203]. These models, beyond ranking among the top-performing models [93], are also widely adopted for code generation tasks [23]. Although our selection focuses on small open-source models, these models demonstrate performance comparable to proprietary ones, surpassing GPT-3.5 [90] and approaching GPT-4 [204], ensuring representativeness in our study. After defining the setup for our study, we proceeded with generating the code snippets. For the hyperparameters, we adopted a conservative approach by setting the temperature to 0, ensuring deterministic outputs and minimizing randomness in code generation. For prompt construction, we followed the default configuration recommended in the official Hugging Face documentation [201–203]. This approach ensured a fair assessment of inherent inefficiencies in LLM-generated code rather than artifacts caused by inadequate prompts. As a result of this step, we have a dataset of 164 generated code snippets for each LLM, resulting in 492 code snippets (B), from the initial dataset of tasks (A).

5.1.2 Step 2: Judging LLM-generated Code

LLMs have been effectively used to assess code or tests [205,206], proposing a paradigm where LLMs serve as judges (LaaJ). Prior studies have also leveraged LLMs to assist in identifying a generated-code bug's root causes [123]. In this context, we propose to leverage LLMs to assess the quality of generated code. Specifically, we employ the *GPT-4o-mini* model, which offers performance comparable to GPT-4 while being more cost-effective [95]. Regarding the hyperparameters, we considered a *temperature* of *0* aiming to achieve a more conservative generation and *max_tokens* set to *5120* to ensure sufficient space for detailed output.

Knowing that *GPT-4* models can be shaped and optimized by setting system prompts, which define the model's persona [207], we adopted the persona of an *expert in code quality analysis* with a focus on identifying specific inefficiencies in machine-generated code. Such a persona guides the model to prioritize practical and significant code inefficiencies rather than minor or irrelevant inefficiencies.

Finally, we defined a fixed user prompt used as input when asking for judgment support (*judger*). In this prompt, we input the generated code snippet and its task description, providing a context to aid the *judger* in identifying logical inefficiencies in the generated code. As a result of this step, we enriched the dataset B with GPT-generated judgments, creating dataset C, as denoted in Figure 5.1. Specifically, C consists of 492 generated code samples, each paired with its corresponding LLM-generated judgment.

5.1.3 Step 3 and 4: Manual Labeling

Step 3: Manual Labeling - Pilot Study With no predefined categories from previous studies, our objective is to construct a taxonomy of inefficiencies using an inductive, bottom-up approach through manual analysis of LLM-generated code snippets. To initiate this process, we randomly selected 196 samples C (40%, pilot set), ensuring an equal proportion of cases from each model. The first two authors, a senior Master's student and a Postdoctoral Fellow with experience in Python and SE research, collaboratively analyzed and labeled the pilot set (40% of samples) using open coding [195], a method widely used for taxonomy construction in SE [208–210]. The analysis involved examining the task description, the generated code, code quality judgment, and the execution and testing status (from the unit tests available in HumanEval+). Without predefined assumptions, inefficiencies were iteratively identified and grouped into categories and subcategories. Since LaaJ may hallucinate [206], we treated their judgments as a starting point and independently verified and expanded on the identified inefficiencies. During this process, the labelers assigned brief, descriptive labels to each inefficiency observed in the pilot set. These initial labels were then categorized, forming a hierarchical taxonomy of inefficiency types. This iterative process involved continuously refining the categories as they moved back and forth between individual samples and emerging patterns, ensuring that the taxonomy accurately reflected the inefficiencies underlying the generated code. All tasks in the pilot set were thoroughly discussed, and the labelers reached a consensus on both categories and subcategories, resulting in a taxonomy comprising 5 inefficiency categories and 18 subcategories and a fully labeled pilot set. This step resulted in a codebook of inefficiency patterns, with multi-labeling allowed, since a single code snippet can exhibit multiple types of inefficiencies [127].

Step 4: Taxonomy Refinement & Full Dataset Labeling After labeling the pilot set and constructing the initial codebook of inefficiencies, the labelers independently labeled the remaining 60% samples, following the same procedure. If any generated code could not be categorized within the existing taxonomy, they were marked as *Pending* and reviewed to

determine whether new categories or subcategories were needed. Through these discussions, *Syntax Error* was added as a subcategory under *Errors*, leading to a refined taxonomy comprising five categories and 19 subcategories, providing a more comprehensive representation of inefficiencies in LLM-generated code.
Once all samples were labeled, we assessed inter-rater agreement to evaluate the consistency between the two labelers. Following similar studies [127, 208], we used Cohen's Kappa [211], calculated at category and subcategory levels. Inter-rater agreement (Cohen's Kappa) reached 0.846 (for category) and 0.735 (for subcategory), aligning with prior SE studies [127, 208].

5.1.4 Step 5: Participants' feedback and survey analysis

To evaluate the relevance and gather insights into the perceived popularity of the identified inefficiencies in real-world settings, we conducted a survey targeting software practitioners and researchers using LLMs in their code generation tasks. Following prior studies [125, 127], we collected the email addresses of GitHub users who collaborated on repositories containing LLM-generated code. To identify relevant repositories, we searched GitHub for repositories where at least one file contained code generated by LLM-based tools or models, including Copilot, ChatGPT, Codex, CodeLlama, Llama, and Llama-2. Specifically, we used the following keyword-based queries: association between `by/with` and `Copilot, ChatGPT, Codex, CodeLlama, Llama, Llama-2`. This process yielded 411 repositories. Next, we extracted publicly available email addresses of contributors from these repositories using PyDriller [212]. This resulted in 835 unique email addresses, of which 51 were unreachable, leaving us with 784 successfully delivered emails (mined on November 18, 2024).
To ensure clear communication and maximize participant engagement, we structured the survey to clearly outline its objectives, structure, and expectations. First, the survey outlined a detailed message about its purpose, scope, and estimated completion time. The survey questionnaire is divided into two main parts. The first part consists of demographic and open-ended questions regarding the LLMs participants have used, as well as the inefficiencies they encountered based on their experiences. The second part assessed the frequency and relevance of inefficiencies in LLM-generated code, as defined in Step 4. For each inefficiency category, we provided a general description, followed by its specific subcategories, each accompanied by a description and an example code snippet. Participants were then asked two questions per subcategory, rated on a 5-point Likert scale: (i) how frequently they encountered the inefficiency reported and (ii) how important they believed it was to address it. Additionally, the survey included an optional question, allowing participants to report any inefficiency categories or subcategories not covered in the taxonomy, based on their experience. The survey remained open for 4 weeks, and we sent a reminder email two weeks after the first

call. Once the results for each inefficiency pattern were collected, we processed the results to evaluate their popularity and relevance. This was achieved by aggregating the results using weighted averages of Likert scale values, following methodologies from similar studies [127]. Additionally, we analyzed participants' feedback to gain deeper insights into their experiences with LLM-generated code. Their comments provided context for interpreting the results and helped identify potential gaps or overlooked aspects in the taxonomy.

5.2 Results

In this section, we present the findings for our two RQs explored in this study. We first introduce our taxonomy of inefficiencies in LLM-generated code, followed by an analysis of the perceptions of practitioners and researchers on the proposed taxonomy, as gathered through our online survey.

5.2.1 RQ1: Patterns of Inefficiencies in LLM-generated Code

Taxonomy

We organized our observations as a taxonomy of five categories with 19 subcategories. The complete taxonomy is shown in Figure 5.2. In the following, we explain each of the five categories with their respective subcategories. Examples of each inefficiency subcategory (i.e., code snippets) can be found in our replication package.

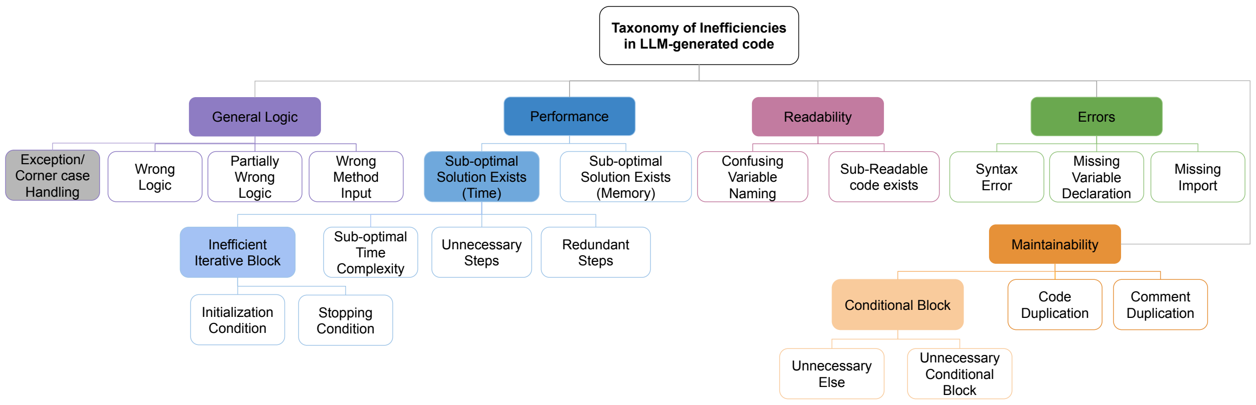


Figure 5.2 LLM-generated Code Inefficiencies: A comprehensive Taxonomy

A. General Logic. This category encompasses inefficiencies related to the general logic of a given task, including inefficiencies in understanding requirements and translating them into code.

- **Wrong Logic.** It occurs when the generated code lacks meaningful logic, containing only comments, hard-coded values, returning method inputs, or using a simple `pass` statement. It also includes case-by-case implementations that do not follow a coherent logic. Additionally, it encompasses situations where all requirements are implemented incorrectly, with none being addressed properly.

- **Partially Wrong Logic** Different from *Wrong Logic*, this subcategory occurs when some requirements requested in the prompt are implemented correctly while others are either implemented incorrectly or not implemented at all.
- **Wrong Method Input.** It occurs when the LLM generates a statement that is generally coherent with the logic but uses incorrect or inappropriate inputs. For example, passing a single element to the `sorted` method instead of a list may lead to an unintended behavior.
- **Exception/Corner Case Handling.** This occurs when the generated code lacks exception handling or fails to address corner cases. Initially, this subcategory was merged with *Partially Wrong*. However, following the suggestions of the survey participants, it was detached as a standalone subcategory.

B. Performance. This category encompasses inefficiencies that degrade the efficiency of the generated code, affecting aspects such as memory management and execution time.

- **Sub-Optimal Solution Exists (Memory)** This subcategory captures cases where the same logic can be implemented in a more memory-efficient way. For example, recursion may be avoided when intermediate results are stored on the call stack, potentially leading to memory overflow for deep recursions. Similarly, list comprehensions, which store all results in memory, can be replaced with generator expressions to enable lazy evaluation and reduce memory usage.
- **Sub-Optimal Solution Exists (Time)** It identifies the cases where there is an equivalent, more time-efficient algorithm compared to the LLM-generated one. In this subcategory, we report four additional subcategories:
  - *Sub-Optimal Solution (Time Complexity)*: It occurs when the implemented logic can be replaced with another algorithm with lower time complexity. For instance, using *Quick Sort*, with a time complexity of $O(n \log n)$ on average, is generally more efficient than *Insertion Sort*, which has a time complexity of $O(n^2)$ in the average and worst cases —particularly when sorting large lists.
  - *Unnecessary Steps.* It refers to instructions in the code that do not affect or contribute to the final output. Eliminating such steps enhances the code's overall efficiency without altering its functionality, such as unnecessary type conversion.

– *Redundant Steps.* This occurs when an essential instruction to the logic is repeated unnecessarily. To avoid such cases, the given instruction could be executed once, and its output could be stored in a local variable instead of re-executing the same instruction.

– *Inefficient Iterative Block.* This category focuses on inefficiencies within repetitive structures, such as loops and iteration management.

+ *Iterative Block (Initialization Condition).* It occurs when the initialization condition of a loop is too broad and can be narrowed without affecting the correctness of the method. For example, when determining whether a number is prime, we can stop checking the divisibility at $\sqrt{n}$ rather than iterating all the way up to n.

+ *Iterative Block (Stopping Condition).* It occurs when the stopping condition for a repetitive block is very broad and can be optimized without changing the intended functionality or when it lacks an early stopping condition when applicable. For instance, when searching for an element in a sorted ascending list, instead of iterating through the entire list, we can stop early if the current element exceeds the target value.

C. Readability This category encompasses inefficiencies related to the overall readability, clarity, and ease of comprehensibility of LLM-generated code snippets.

- **Confusing Variable Naming.** It refers to variable names that may confuse the developer, even if they do not result in runtime errors. It includes inefficiencies such as using existing identifiers to define new variables (e.g., using `dict` as a variable or method name) or variable shadowing, where the same variable name is used in different scopes.

- **Sub-readable method exists.** This refers to cases where the code is difficult to understand or does not adhere to conventions. For example, the combination of list comprehension, `map`, and `lambda` functions, along with a for `loop` and multiple method calls, such as the use of this statement: `return reduce(lambda x, y:  x * y, [int(i) for i in str(n) if int(i) % 2 == 1])`.

D. Maintainability. This category highlights inefficiencies in LLM-generated code that may negatively impact the maintainability of the code, possibly intended to be integrated into larger systems.

- **Code Duplication.** This occurs when identical or similar blocks of code appear in multiple places without contributing to the main logic, essentially resembling code cloning [213].

- **Comment Duplication.** It occurs when repetitive comments are used excessively, identical, or similar, without providing additional insight or explanations about the logic.

- **Conditional Block.** It groups inefficiencies related to conditional blocks, i.e., `if` blocks. Although it does not affect the functionality, it is important to be addressed for better code clarity and easier testing/debugging [214]. While in our studied sample we did not observe `foreach` blocks, this subcategory may also be extended to cover them.

  – *Unnecessary Else.* It takes place when an `else` statement is used unnecessarily within an `if` block and can be removed without affecting the logic. It usually happens when the preceding `if` or `elif` statements already contain a `return` or `break`, making the `else` unnecessary.

  – *Unnecessary Conditional Block.* It refers to cases where using a conditional block adds unnecessary complexity, and the code can be simplified by directly using the output of a given condition in a `return` statement.

E. Errors Inefficiencies falling under *Erros* are related to the fundamental correctness of the code regardless of the logic being implemented, like runtime errors, unexpected behavior, or failure to execute the code properly.

- **Missing Import.** It occurs when methods or constants from external modules/libraries are used without being properly imported. Missing imports can lead to errors or failures when the code attempts to use the functionality provided by those modules.

- **Missing Variable Declaration.** It occurs when a variable is used without being properly declared, initialized, or passed as an argument for a method, which may result in runtime errors.

- **Syntax Error.** This occurs when the code contains a syntax error preventing it from being executed, such as improper use of keywords or incorrect indentations in Python.

Overall, LLM-generated code exhibits similar inefficiencies to human-written code. While LLMs often pass more test cases than human developers, they struggle with tasks requiring domain expertise [41]. Regarding performance, although LLMs can generate efficient solutions for certain tasks, they often under-perform compared to human-optimized implementations [126]. In terms of readability, LLM-generated code is generally comparable to human-written code but may deviate from coding standards [40, 215]. However, its increased complexity can elevate maintenance effort [41]. While LLMs can address maintainability issues, their fixes frequently introduce new errors at a higher rate than human corrections [216].

Analysis of Inefficiency Patterns

Table 5.1 (1) Frequency of inefficiencies in the studied sample of LLM-generated code, and (2) Survey-based taxonomy validation Results

	(1)Frequency in of ineffeciencies in LLM-generated code(%)				***(2) Survey-based Validation (Weighted average)***	
	Codellama	*Deeepseek-Coder*	*CodeGemma*	*Overall*	*Popularity*	*Relevance*
1- General Logic	60.37	70.73	**74.39**	68.5	—	4.03
Wrong Logic	9.76	39.63	**57.93**	**35.77**	2.22	3.64
Partially Wrong Logic	**47.56**	27.44	15.24	30.08	3.17	3.88
Wrong Method Input	**4.27**	3.66	1.22	3.05	2.76	3.53
2- Performance	**48.78**	30.49	23.17	34.15	—	3.52
Sub-Optimal Solution (Memory)	**26.83**	17.07	12.2	**18.7**	3.07	3.12
Iterative Block (Initialization Condition)	0.61	0.61	0.61	0.61	2.69	3.19
Iterative Block (Stopping Condition)	**3.66**	1.22	2.44	2.44		
Redundant Steps	**9.76**	3.05	3.05	5.28	2.64	3.05
Unnecessary Steps	**4.27**	2.44	0.61	2.44	3.03	3.03
Sub-Optimal Time Complexity	**23.78**	17.68	14.02	18.5	2.93	3.12
3-Readability	**6.71**	6.1	1.22	4.67	—	3.22
Confusing Variable Naming	1.22	1.22	0.61	1.02	2.17	3
Sub-Readable Code Exists	**5.49**	4.88	0.61	**3.66**	2.79	3.05
4-Maintainability	**29.88**	12.8	20.73	21.14	—	3.31
Unnecessary Else	**21.95**	10.98	19.51	**17.48**	2.29	2.59
Unnecessary Conditional Block	3.66	1.83	3.66	3.05	2.62	2.78
Code Duplication	**4.88**	1.83	0.61	2.44	2.46	3.02
Comment Duplication	**3.05**	0	0.61	1.22	1.86	2.10
5-Errors	**7.32**	6.1	3.66	5.69	—	3.93
Syntax Error	0	2.44	2.44	1.63	2.29	3.79
Missing Variable Declaration	0.61	0	0.61	0.41	2.17	3.60
Missing Module Import	**6.1**	3.66	1.22	**3.66**	2.86	3.26

Most samples were labeled with a single category (56.5%), followed by those with two (24.59%) and 9.96% were not labeled, as raters found no inefficiencies and considered them efficient. A similar distribution appeared at the subcategory level with 54.07% of samples assigned one subcategory followed by 18.29% with two and 11.38% with three. Notably, 33.54% of the samples spanned multiple categories, highlighting the interconnected nature of inefficiency patterns across different code quality aspects.

Table 5.1 presents the frequency of inefficiency categories and subcategories (since multi-labeling is allowed, totals may exceed 100%). *General Logic* emerges as the most frequent category across all models (68.5% of samples). Among the selected models, CodeGemma, the least efficient, based on Top@k (leaderboard [93]), exhibited the highest rate of logic-related inefficiencies (74.39%), often generating code without a meaningful logic (57.93%). While the most efficient model based on Top@K, CodeLlama, showed fewer critical logic errors (60.37%), though it also frequently generated partially correct solutions (47.56%), suggesting that even advanced models may misinterpret requirements, which is aligned with findings of other research works [217].

Performance inefficiencies ranked second (34.15%), with CodeLlama exhibiting the highest frequency within this category. *Sub-Optimal (Memory)* (18.7%), *Sub-Optimal (Time Complexity)* (18.5%) and *Redundant Steps* (5.28%) were the most prevalent performance-related inefficiencies. These inefficiencies are not exclusive to smaller or open-source models; prior studies have reported similar performance issues in GPT-4-generated code [133], reinforcing the broader challenge of optimizing LLM-generated code across different model architectures.

Maintainability inefficiencies varied significantly between models. DeepSeek-Coder, trained exclusively on code, exhibited the lowest *Maintainability* issues (12.8%), whereas CodeLlama and CodeGemma (trained on mixed code/natural language data) have higher rates (29.88% and 20.73%, respectively). This discrepancy suggests that code-specific training may improve structural quality, while incorporating natural language data in training may dilute the focus on maintainability. The prevalence of *Unnecessary Else* blocks (21.95% in CodeLlama) underscores a broader challenge: LLMs often generate unnecessarily complex control flows, which complicate long-term maintenance [218].

Readability (4.67%) and *Errors* (5.69%) were less frequent but non-trivial. *Syntax Errors* were rare ($< 2.44\%$), confirming the strong syntactic grasp of code LLMs. However, Missing Imports (6.1% in CodeLlama) reveal persistent gaps in dependency resolution, a known limitation even in highly-efficient models like GPT-4 [219].

To gain deeper insights into inefficiency relationships, we analyze how different categories co-occur across models. Figure 5.3 presents the normalized category co-occurrence heatmap across all models. The co-occurrence values are normalized row-wise by dividing each occurrence by the frequency of the corresponding category. Our results reveal strong interdependencies between inefficiency patterns in LLM-generated code. *General Logic* inefficiencies exhibit the highest co-occurrence rates with other categories, particularly *Readability (0.74)*, *Errors (0.64)*, and *Maintainability (0.58)*. This suggests that flawed logic not only impacts correctness but also contributes to inefficiencies in clarity, structure, and long-term maintainability of the generated code. Similarly, *Performance* inefficiencies frequently co-occur with *Readability (0.48)* and *Maintainability (0.46)*, highlighting that performance inefficiencies are often tied to code complexity and structure, rather than isolated algorithmic inefficiencies. Interestingly, *Errors* demonstrate weaker co-occurrences with other inefficiencies, with their strongest link being to *Maintainability (0.36)*, indicating that while syntax and dependency errors are less common, they may still hinder maintainability efforts by requiring additional debugging and refactoring. Meanwhile, *Readability* and *Maintainability* are closely related (0.35), reinforcing the idea that clear code structures contribute to long-term main-

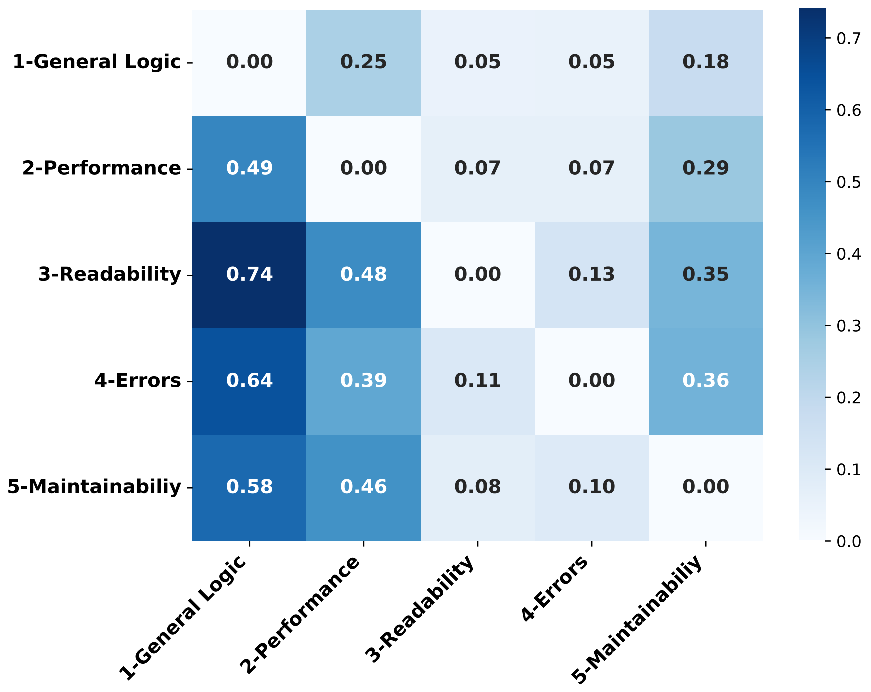


Figure 5.3 Normalized Category Co-occurrence

tainability. These findings suggest that improving LLM-generated logic and performance could lead to broader improvements across multiple inefficiency categories, reducing the need for manual corrections and enhancing code usability in real-world development.

5.2.2 RQ2: Relevance of Identified Inefficiencies for Practitioners and Researchers

The survey remained open for four weeks, reaching 58 responses, resulting in a response rate of 7.39%, which is in line with rates observed in similar surveys in Software Engineering [208, 220–222]. First, we present the demographic information about the participants and their experience with LLM-assisted coding. Then, we provide a detailed analysis of the popularity and perceived relevance of inefficiencies in LLM-generated code reported by the participants.

Demographic Information and Experience with Coding using LLMs

All participants answered the demographic questions. We had participants from both academia (60%) and industry (40%). The academic participants consisted of graduate students (29), undergraduate students (3), and postdoctoral researchers/lecturers (3). Industry participants included 11 software developers, 7 ML/data science developers, 4 senior executives, and 1 cybersecurity developer. About coding experience, 37 participants had more than 5 years of experience, while 11 had between 3-5 years, and 10 had between 1-3 years. LLM-assisted

coding is widely adopted, with 98% of the participants using LLMs for coding. Among them, 68% rely exclusively on proprietary models, with ChatGPT and Copilot being the most popular (89.66% and 29.31%, respectively). Regarding open-source, the most popular models are Mistral, the Phi model family, and Qwen-Coder (5.17%, 3.45%, and 3.45%, respectively). Participants reported using a variety of programming languages, with the top five being Python, C/C++, JavaScript, Java, and Go (91.38%, 18.97%, 18.97%, 8.62%, and 3.45%, respectively).
Before the questions about our taxonomy, we asked participants about their general experiences and challenges with LLM-generated code. Reported challenges can be categorized into two main groups: *user experience* (outside the scope of our taxonomy) and *code-related*, which are directly aligned with our research. We discuss the inefficiencies encountered by the participants based on their experiences before seeing the taxonomy.

User Experience-Related Inefficiencies These inefficiencies focus on the challenges and difficulties that users face when interacting with LLMs, rather than the generated code itself. **6** participants reported that LLMs struggle with debugging or modifying specific parts of code, a finding consistent with previous studies [223]. In the same way, programming language and library popularity also play a significant role in the efficiency of LLM-generated code. **5** participants noted that LLMs generate more efficient code for popular languages and libraries like Python and NumPy compared to less common ones like DHL and Tinkter. This difference is likely because LLMs are trained on more examples of popular languages and libraries, making them more familiar with their syntax and usage. Programming language paradigms also play a role—P12[1] observed that LLMs handle interpreted languages better than compiled ones. Finally, participants expressed varying preferences for chat-based LLMs (e.g., ChatGPT) versus inline code assistants (e.g., Copilot). While P41 found Copilot suggestions as "stupid suggestions and distracted", P11 preferred inline assistance over chat-based interactions.

Code-Related Inefficiencies Participants also reported various inefficiencies in the generated code that align with the focus of our taxonomy. **19** participants stated that LLMs often struggle to produce the correct logic on the first attempt, even for simple tasks with detailed descriptions, requiring multiple further iterations. P22 specifically noted that LLMs may generate "unnecessarily long or complex code (for simple tasks)" However, P15 found that for more complex tasks such as building a Flask application, LLMs tend to perform "over-simplification to the point of uselessness", where generation is limited to high-level steps or

[1]We refer to survey participants using the notation Px, where x is an assigned participant ID.

generalized code that lack specificity, performance efficiency, and functionality. Additionally, **3** participants reported that handling exceptions and corner cases is often overlooked. These reported inefficiencies fall under the *General Logic* category of our taxonomy.

Another notable challenge reported by the participants is the misalignment between LLMs' advancements and technological progress. **13** participants reported that LLMs struggle with version tracking, frequently generating code that uses deprecated methods, mixes library versions, or is incompatible with specified APIs (also mentioned by previous works [219]). These inefficiencies align with *Wrong Method Input* (*General Logic*) and *Missing Import* (*Errors*) subcategories of our taxonomy.

Readability inefficiencies were also reported by 7 respondents spanning various programming languages, like Python, C/C++, Java, JavaScript, Go, and Bash. They highlighted poor formatting, failure to adhere to coding conventions, inconsistent variable names, and reliance on outdated practices as reported by Zhang et al. [40].

Additionally, **6** participants highlighted performance issues. Others pointed to a lack of robust, modular, object-oriented architecture and insufficient documentation and comments as well. The inefficiencies reported by the participants align with the literature [40, 40, 123] and fall under *Readability*, *Performance*, and *Maintainability* categories of our taxonomy.

Besides inefficiencies, **7** participants identified context limitations as a key issue. **2** participants found that LLMs had a limited context window and frequently forgot past interactions, as mentioned in related work [126]. Meanwhile, P5 reported that LLMs tended to retain outdated context, stating that they needed to "start a new chat to get rid of hallucinations."

Popularity and Relevance of Code Inefficiencies: Survey Perception

Table 5.1 (6th and 7th columns) summarizes the aggregated results of popularity and relevance of the subcategories of code inefficiency patterns on a scale of 1 to 5. The *popularity* score reflects how frequently participants faced each type of inefficiency, while *relevance* aims to indicate whether the participants consider that such an inefficiency should be prioritized. Results show that the frequency of inefficiencies in the studied sample set is aligned with popularity scores reported by participants.

General Logic inefficiencies were rated as the most relevant category (4.03/5). In particular, *Partially Wrong Logic* stands out as the most relevant and popular subcategory encountered by participants (3.88/5 and 3.17/5, respectively) and is highly frequent in our studied sample set (30.08%). This observation suggests that while LLMs generate code that often aligns with task descriptions, they frequently miss certain requirements, requiring manual effort to refine the logic. Similar observations were also reported in other studies, particularly those analyzing Copilot-generated code [42]. Although *Wrong Logic* was reported as the most frequent subcategory in our studied sample (35.77%), it receives a relatively low popularity score (2.22/5), which can be explained by the fact that participants mostly use proprietary models, with 98% relying on at least one such model. Since proprietary models generally perform better than open-sourced ones, participants may encounter *Wrong Logic* less frequently in their own experience.

Performance inefficiencies, with a frequency of 34.15% in our studied sample, were also considered highly relevant (3.52/5) and widely encountered by participants with a popularity score (3.0/5). The most concerning inefficiencies were *Sub-Optimal (Memory)* and *Sub-Optimal (Time Complexity)* (3.0/5 for popularity, 3.12/5 for relevance), as well as *Inefficient Iterative Blocks*, which directly impact execution time.

This finding suggests that while correctness is prioritized, performance inefficiencies in LLM-generated code can still add to developers' workload, as highlighted by other researchers [42].

For *Maintainability*, participants reported a moderate popularity (2.3/5) but a high relevance score (3.31/5), indicating that while maintainability inefficiencies are encountered less frequently, respondents still considered them important. *Unnecessary Else*, the most frequently observed maintainability inefficiency in our studied sample (17.48%), had the lowest relevance score (2.59/5), as many developers considered it a minor issue that does not affect correctness. In contrast, *Code Duplication*, despite being the least frequent inefficiency (2.44%) and less popular (2.46/5), was rated more relevant (3.02/5) due to its impact on maintenance effort and complexity of refactoring. This suggests that developers prioritize maintainability inefficiencies that directly increase long-term maintenance costs, even if they occur less often.

Readability inefficiencies were relatively rare (4.67% frequency) but still perceived as moderately relevant (3.22/5.0) despite low popularity scores. This observation highlights that even infrequent readability inefficiencies are critical to be addressed, as developers typically resolve them manually rather than relying fully on automated fixes [224]. Finally, while *Errors* were the least frequent (5.69%) and the least popular (2.25/5) inefficiencies encountered by participants, they had a high relevance score (3.93/5.0). The participants emphasized that, although rare, these errors must be addressed, as they prevent the code from being compiled or executed.

Our findings suggest that the inefficiencies identified in our analysis accurately reflect the challenges practitioners and researchers encounter when dealing with LLM-generated code.

LLM-generated Code Inefficiencies Taxonomy: Feedback and Completeness

We gathered participants' feedback through open questions for this part. Regarding *General Logic*, participants noted that *Wrong Logic* often occurs when tasks are complex or involve new/unfamiliar technologies. For *Partially Wrong Logic*, participants acknowledged that they often correct these inefficiencies but emphasized that LLMs sometimes *"generate buggy solutions, and debugging takes longer than writing the code from scratch"*(P41) Additionally, some participants pointed out that partially correct logic can be difficult to detect, increasing the risk of unnoticed errors at runtime. P2 was concerned that: *"A junior coder who uses this could cause issues if pushed to production code without proper review"*. For *Performance*, participants from both academia and industry noted that they do not rely on LLMs for generating highly optimized code. Instead, they prioritize correctness, ensuring the code is bug-free, and manually optimize performance only when necessary. Regarding *Readability*, participants were indifferent to the clarity and structure as long as the code functioned correctly and could be manually refined if needed. For *Maintainability*, participants noted that inefficiencies like "code and comment duplication" are less noticeable when using tools such as Copilot [225] and Cursor AI [226]. *Unnecessary Else* statements were considered low-impact, and participants showed little interest in addressing them. For *Errors*, while they can lead to runtime failures, participants found them easy to detect using IDEs and correct.

Participants acknowledged that prompting can help mitigate inefficiencies related to *General Logic*, *Performance*, and *Readability.* However, certain inefficiencies, such as *Wrong Method Input*, still require developer expertise to resolve effectively. Overall, participants agreed that the proposed taxonomy effectively captures inefficiencies in LLM-generated code, evidenced by its high popularity and relevance scores. Many of these inefficiencies were already mentioned by participants before they were introduced to our taxonomy (5.2.2), demonstrating a strong coverage. However, three participants suggested introducing a new subcategory for *Exception and Corner Case Handling*, as these inefficiencies were not explicitly included in our taxonomy.

5.3 Threats to Validity

Construct Validity. The process of collecting and labeling generated code may influence our study's results and findings. While our methodology aligns with existing works proposing taxonomies in SE [127, 208], we mitigate such a risk by clearly explaining our approach in this paper to ensure external validation. The use of LaaJ introduces a potential threat, as LLMs may produce incorrect or inconsistent judgments. To address this risk, we treated the LLM's judgments as an initial reference rather than a definitive verdict. The two labelers independently reviewed and verified all judgments to minimize inaccuracies and mitigate hallucinations. In the absence of LLM-generated code inefficiency categorization, we recognized the risk of introducing bias into our classification. To address this, we adopted an open coding procedure and utilized different LLM models to enhance the robustness and reliability of our findings. Additionally, we surveyed practitioners and researchers to validate the completeness of our taxonomy.

Internal Validity. A key internal threat to validity is the potential bias in the manual labeling of LLM-generated code. To mitigate such a threat, the first two authors independently labeled the generated code snippets. The inter-rater agreement at the category level is 0.846, which aligns with similar studies in software engineering [127, 208]. Additionally, to further validate our findings, we surveyed 58 participants from both academia and industry, ensuring a broader perspective on the identified inefficiencies. Additionally, we leveraged an LaaJ to assess generated code snippets, providing an additional layer of validation and reducing potential human bias in the labeling process. However, we recognize that LaaJ judgments may have missed certain inefficiencies or introduced biases that could influence human labeling. To mitigate this risk, the two labelers independently reviewed, refined, and expanded upon the LLM's judgments, ensuring that inefficiencies were identified beyond those initially flagged by the model.

External Validity. One external threat to validity concerns the selection of LLMs used to generate code. In this study, we utilize three open-source LLMs: CodeLlama, DeepSeek-Coder, and CodeGemma, all of which have demonstrated efficiency and have been employed in prior studies to leverage the capabilities of LLMs for code generation [227–229] Another potential threat stems from the complexity of the benchmark tasks, as they may not fully represent real-world programming scenarios, which could limit the generalizability of the inefficiencies identified. To mitigate this threat, we use the HumanEval+ dataset, a widely adopted benchmark for evaluating LLM-generated code [198–200]. Such a benchmark consists of human-crafted algorithmic tasks, which, while not encompassing the full breadth of software development tasks, represent a fundamental component of real-world programming. Furthermore, our study focuses exclusively on Python code, the most widely used programming language. However, this scope introduces a limitation, as we may have overlooked inefficiencies that are specific to other programming languages. Future work could extend our analysis to additional programming languages to enhance generalizability. Another critical threat is the long-term relevance of our proposed taxonomy, given the rapid evolution of LLMs. As these models continue to improve, some inefficiencies may become less prevalent while new ones may emerge. To address this threat, future research should periodically reassess and refine the taxonomy to ensure its continuous applicability in evaluating LLM-generated code.

Conclusion Validity. Some types of inefficiencies may have been overlooked, potentially affecting the conclusions of this paper. To mitigate this threat, we manually inspected 492 code samples generated by three different LLM models. To minimize errors, two annotators independently labeled each sample, followed by a discussion to resolve discrepancies. Finally, the results were validated through a survey. We also provide a replication package to support the reproducibility of our findings [196].

5.4 Chapter Summary

In this chapter, we systematically investigated inefficiencies in LLM-generated code and introduced a taxonomy that organizes these issues into five categories: *General Logic*, *Performance*, *Readability*, *Maintainability*, and *Errors*, encompassing 19 specific subcategories. We validated the taxonomy through a survey involving 58 practitioners and researchers, who largely affirmed its completeness and the real-world relevance of the identified inefficiencies. Our findings reveal a critical gap in the ability of LLMs to consistently generate efficient and high-quality code. *General Logic* and *Performance* inefficiencies emerged as the most prevalent, often appearing alongside *Maintainability* and *Readability* issues—highlighting the interconnected nature of quality concerns in LLM-generated code. By offering a structured understanding of these inefficiencies, the proposed taxonomy supports more trustworthy evaluation of LLM outputs and enables targeted model improvements. More importantly, it contributes to enhancing the reliability and trustworthiness of LLMOps pipelines by laying the groundwork for systematic, efficiency-aware code assessments—ensuring that code deployed through these pipelines meets not only correctness requirements but also broader software quality standards.

In the following chapter, we build on this taxonomy to design and implement a regression testing framework aimed at detecting quality regressions in LLM-generated code. This framework operationalizes the identified inefficiencies, allowing for automated and repeatable assessments of code quality across LLM versions—further strengthening the reliability of LLMOps workflows.

CHAPTER 6 TOWARDS RELIABLE LLMOPS PIPELINE FOR CODE GENERATION: A REGRESSION TESTING APPROACH

With the increasing capabilities of LLMs in code generation [37], new models are rapidly emerging—from proprietary models like GPT-4 to open-source alternatives such as CodeLlama [88] and DeepSeek-Coder [90]. These models are also frequently adapted using techniques like fine-tuning and model merging. Fine-tuning enhances model specialization by training on domain-specific data [230], as demonstrated by Li et al., who fine-tuned GPT-J to improve C/C++ proficiency. Model merging, on the other hand, integrates complementary capabilities, such as mathematical reasoning and logical coherence, to boost code generation performance [141].

However, as these models evolve, the reliability and trustworthiness of LLM-generated code remain a critical concern in LLMOps. Although LLM-generated code often passes functional tests, it frequently suffers from inefficiencies related to maintainability, performance, readability, and security [40,42,124,125,231]. Yet, most existing evaluation frameworks prioritize functional correctness [23], neglecting broader quality attributes essential for real-world deployment [40].

As LLMs continue to be released, ensuring that newer versions do not introduce or exacerbate such inefficiencies becomes an urgent need for reliable and trustworthy LLMOps pipeline underlining the need for robust regression testing. Unlike traditional regression testing in SE—which relies on re-executing deterministic unit tests [137]—LLM regression testing faces challenges due to the models' nondeterminism. A single prompt may yield varied outputs, complicating expected result definitions and regression detection [138].

While efforts in LLMs regression testing are limited, RETAIN [139] was designed as a general-purpose LLM regression testing framework, focusing primarily on textual differences between LLMs outputs. However, textual differences alone cannot effectively assess code generation because code is not just text; it has structural, semantic, and executional properties that impact both functional and non-functional aspects [44]. This limitation makes RETAIN a non-suitable regression testing framework for code generation.

To address this gap, we propose *ReCatcher*, to the best of our knowledge, the first LLM regression testing framework for code generation. *ReCatcher* systematically compares the code generation capability of two LLMs—the current model in use and a potential model for update—and generates a comprehensive regression report. Our framework tests regression across three key aspects: logical correctness, performance, and static code issues (including readability, maintainability, and errors), leveraging a taxonomy of inefficiencies in LLM-generated code [232] reinforcing trustworthiness and reliability of within LLMOps pipelines for code generation. *ReCatcher* integrates widely validated software testing tools: unit tests for correctness [233], static analysis for static code issues [234, 235], and profiling tools for performance [236]. By systematically applying these established methods and comparing LLM-generated code snippets across models, ReCatcher provides a robust and reproducible assessment of regressions in LLM-generated code. *ReCatcher* supports specifically Python, considering its popularity and wide adoption [197].

We used *ReCatcher* to assess the impact of three common model update scenarios on LLMs code generation capability: fine-tuning [104], merging [98], and model release which involves introducing a new version within a model family (e.g., the release of GPT-4o-mini [95]). Our assessment covers both open-source and proprietary LLMs. For open-source models, we tested regression when updating to the fine-tuned and merged variants of the widely used CodeLlama [89] and DeepSeek-Coder [90]. The selected variants were retrieved from Hugging Face [237], prioritizing those with the highest number of downloads. For proprietary models, we assessed regression when updating from GPT-3.5-turbo to GPT-4o and from GPT-4o to GPT-4o-mini. The GPT family is among the most widely used model families for code generation [23]. To ensure a comprehensive evaluation, we used two benchmarks: HumanEval+ [37] (which consists of algorithmic problem-solving tasks) and BigCodeBench [131] (which covers real-world software development tasks).

Across all tested scenarios, the most prevalent regressions occurred in logical correctness, errors (syntax error and missing declaration/import), and performance, while readability and maintainability were relatively stable. Based on our results, we recommend prioritizing testing theses prevalent aspects before model adoption. Regarding the different adaptation techniques, we observe that fine-tuning models on datasets from different programming languages led to syntax errors, resulting in a regression of up to 12%. We observe that LLMs merging can either mitigate or amplify regressions depending on the merged model specialization and training objective. For instance, merging with general-purpose LLM (a non-coding optimized LLM such as Llama2 [88]) introduced regression across various code quality aspects, such as logical correctness, by up to 18%. Finally, regarding model releases, GPT-4o exhibited regression in handling missing declaration/import compared to GPT-3.5-turbo up to 50%. Additionally, GPT-4o-mini showed performance regressions compared to GPT-4o; specifically, it regressed by 80% for execution time. For merging, selecting models with coding capabilities helps mitigate regressions. Fine-tuning on datasets from different programming languages increases syntax errors. These findings highlight the importance of systematic regression testing to ensure reliable, high-quality updates in LLMOps workflows. We recommend prioritizing evaluations of logical correctness, performance, and static errors

before deploying updated models in production environments.
To summarize, in this study, we make the following contributions towards enhancing reliable and trustworthy LLMOps pipelines::

- We propose *ReCatcher*, the first systematic LLM regression testing framework for Python code generation across multiple code quality aspects.
- We use our proposed framework to assess the impact of model updates across three key scenarios: fine-tuning, merging, and new model release.
- We open-source *ReCatcher*, providing a reusable and extensible regression testing framework to support informed LLM updating decisions [238].

The remainder of this chapter is organized as follows: Section 6.1 introduces the efficiency tests used for assessing LLM-generated Python code. Section 6.3 details how *ReCatcher* integrates these tests to systematically assess regressions; presenting its main concepts and workflow. Section 6.3 describes the experimental setup and analyzes the results. Section 6.5 discusses threats to validity.

6.1 Automating LLMs Code Generation Capability Assessment

To assess LLMs' code generation capability, we focus on the generated code, as a more capable LLM should generate more efficient code. This approach aligns with existing methods, such as CodeBLEU and pass@k [23], which also evaluate LLMs based on generated code. Directly assessing code efficiency is challenging due to its multi-faceted nature. To address this, we take an inverse approach, identifying inefficiencies as a proxy for efficiency, mirroring two sides of the same coin relationship between correctness and incorrectness [239]. By systematically detecting inefficiencies, we provide a structured way to assess LLM-generated code beyond functional correctness, uncovering critical quality aspects that traditional evaluating approaches overlook.
Prior studies have attempted to categorize LLM-generated code inefficiencies. For instance, [123, 127] focused on buggy code, grouping inefficiencies as underlying bug causes. However, inefficiencies are not limited to correctness errors. Moradi et al. [42] highlighted that even syntactically and semantically correct code can still suffer from inefficiencies affecting readability, maintainability, and performance. Building on this, our study takes a broader perspective on inefficiencies in LLM-generated code, considering issues regardless of correctness. To systematically assess these inefficiencies, we rely on a structured taxonomy [232] serving as foundation for developing dedicated efficiency tests, enabling a comprehensive evaluation

of LLM-generated code quality. The used taxonomy groups inefficiencies into five main categories *General Logic*, *Performance*, *Maintainability*, *Readability*, and *Errors* further divided into 19 sub-categories [232]. To ensure a structured assessment, we group these inefficiencies into three key aspects: logical correctness, static code issues, and performance. Specifically, logical correctness is mapped to the *General Logic* (*Wrong Logic*, *Partially Wrong Logic*, *Wrong Method Input*, and *Exception/Corner Case Handling*) category. Performance is mapped to the *Performance* subcategories (*Sub-optimal (Memory)* and *Sub-optimal (Time)*). Static code issues include *Errors* (Syntax Error, Missing Declaration/Import), *Readability* (Confusing Variable Naming, Sub-readable Code), and *Maintainability* (Code Duplication, Comment Duplication, Unnecessary Else, Unnecessary Conditional Block).
Inefficiencies are not unique to LLM-generated code, as human-written code is also susceptible to them [240, 241]. Consequently, significant efforts have been made to develop tests for inefficiency detection. Below, we outline existing methods and discuss how we leveraged them to detect inefficiencies in LLM-generated code.

6.1.1 Logical correctness

According to the taxonomy of inefficiencies [232], *General Logic* subcategories share a common characteristic—their impact on logical correctness. The automation of logical correctness assessment has been widely studied, with various approaches proposed in the literature. Formal verification-based methods are particularly notable, employing concepts such as state machines and symbolic execution [242]. However, such methods are inherently complex [243] and involve substantial computational overhead [244]. With the emergence of LLMs, researchers have also leveraged LLMs themselves as automated judges to evaluate code correctness. However, an LLM's ability to generate correct code does not necessarily mean it can accurately assess the correctness of other code [205].
Given these limitations, we opt for a more practical, stable and scalable approach: leveraging unit tests to automate logical correctness assessment. Unit tests execute the code with predefined inputs and compare the actual outputs to expected results. If a given code snippet passes all tests associated with its initial task, it is considered logically correct; otherwise, failures indicate inefficiencies in logic [245]. Despite potential challenges, such as incomplete coverage and weak test suites, unit tests remain an effective tool for validating code correctness, as they can be complemented by generated-tests using tools like EvoSuite [233] or by leveraging LLMs to enhance coverage [233, 246, 247]. In our approach, we do not explicitly automate the detection of all inefficiency subcategories within *General Logic*. Instead, we focus on detecting their consequences—namely through test failures. This proxy-based approach offers a practical solution for assessing logical correctness at a category level in

LLM-generated code.

6.1.2 Static Code Issues

Static code issues are patterns or structures in code that negatively impact software quality across various aspects, including readability, maintainability, and performance [234,248]. Static analysis offers an effective approach for identifying and detecting such inefficiencies without requiring code execution [249]. This approach is fully automated, scalable, while helping developers catch issues in the development process and significantly reducing the time and effort needed for manual code review [250]. Several static analysis tools are available, ranging from general-purpose tools, that address common code issues, to specialized tools focused on specific aspect of code such as security vulnerability detection. These tools can be tailored to specific or support multiple programming languages. For instance, SonarQube [251] provides seamless integration with CI/CD pipelines and supports a wide range of programming languages. Additionally PMD, as a cross programming language tool, includes Copy-Paste Detector (PMD-CPD) for identifying code duplication using configurable rule sets [252]. However, such tools often introduce significant overhead compared to more specialized solutions, such as linters [253]. Linters, typically designed for a single programming language, are easier to integrate into IDEs and allow for custom rule extensions [254]. To ensure scalability and effectiveness, we employ linters for detecting static code issues in LLM-generated code. Specifically, we use Pylint [235], which is regarded as the most popular [255] and efficient Python linter [256]. Pylint analyzes code by constructing an Abstract Syntax Tree (AST) and applying a set of predefined, extensible rules. Detected issues are reported and mapped to specific unique predefined messages [235]. To automate the detection of static inefficiencies from the taxonomy [232], we mapped each inefficiency subcategory to its corresponding Pylint messages by systematically running Pylint on a labeled inefficient code dataset [232] and collecting its outputs. Most inefficiencies were mapped on a one-to-one or one-to-many basis with Pylint messages, where violations of these messages indicate inefficiencies. However, certain inefficiencies required custom handling. For example, *Missing Import* and *Missing Variable Declaration* subcategories from the *Error* category could not be distinguished as Pylint reports both under the same messages for undefined symbols. Consequently, we combined these two subcategories into a single test. Additionally, Pylint does not natively detect *Unnecessary Conditional Block* inefficiency. To address this limitation, we extended Pylint by defining a custom rule that flags patterns matching `if condition: return True|False`. Pylint also lacks native support for code duplication detection. Rather than extending Pylint for such a goal, we opted for an established external tool to ensure accuracy and reliability. PMD-CPD detects structural similarities using token-based approach allowing it to detect duplications even if they slightly differ in terms of variable names or formatting. Given its effectiveness and reliability, we utilized PMD-CPD to automate the detection of code duplication and comment duplication, key subcategories in the taxonomy of inefficiencies [257, 258].

6.1.3 Performance

The *Performance* category in the used taxonomy is organized into several subcategories at multiple levels [232]. In our study, we focus on detecting inefficiencies at top level, without further subdividing them into finer-grained subcategories. Several methods exist for automatically evaluating code performance. Static analysis, while applicable to performance assessment, occasionally yields in incorrect results as performance inefficiencies are difficult to detect without execution, potentially leading to unnecessary code changes [259]. Calotoiu et al. [260] proposed a scalability bug detection approach, while various profiling techniques have emerged [261]. However, these methods are limited to assessing performance at the scalability level. AST-based approaches, such as those used in LeetCode [262], are common in educational settings, though they are less effective for real-world code assessment. Assessing the performance of LLM-generated code is challenging, as it requires determining whether a given solution is optimal in terms of execution time and memory usage. Since our primary goal is to compare the performance of generated-code from two models and detect potential regressions, we adopt a direct performance efficiency comparison approach rather than evaluating a single solution in isolation. Specifically, we execute the LLM-generated code from both models and profile their execution time and memory usage. We measure execution time by recording the start and end times. For memory usage, we leverage tarcemalloc from Python [263], allowing us to track the memory usage through execution. By leveraging this direct comparison approach, we can effectively detect regressions, improvements, or stability in LLM-generated code snippets across execution time and memory usage.

6.2 ReCatcher: LLMs Regression Testing Framework for Code Generation

In this section, we introduce ReCatcher, our LLMs regression testing framework for Python code generation. We begin with an overview of the framework, then we provide a detailed breakdown of its architecture.

6.2.1 Overview

ReCatcher takes as input two LLMs: a deployed actively used model (LLM) and a potential replacement (LLM'), compares and generates a detailed regression report across various code aspects. The framework evaluates the code generation based on a benchmark of programming tasks. ReCatcher follows a three-step process:

- Code Generation, where code snippets are generated for the code tasks reported in the benchmarks, using both LLMs with repeated generation (m times) to mitigate randomness;
- Test Execution, where efficiency tests are executed to assess logical correctness, static issues, and performance. To ensure statistical robustness, performance tests are repeated n times; and
- Analysis, where test outputs are used to generate a comprehensive comparative report comparing code generation capability between LLM and LLM', highlighting improvement and regression.

6.2.2 Architecture

Figure 6.1 illustrates ReCatcher's architecture, which follows a modular, plugin-based architecture providing flexibility and customization for testing LLMs. ReCatcher consists of three main components: Code Generator, Test Suite, and Analyzer. Below, we describe each component in detail.

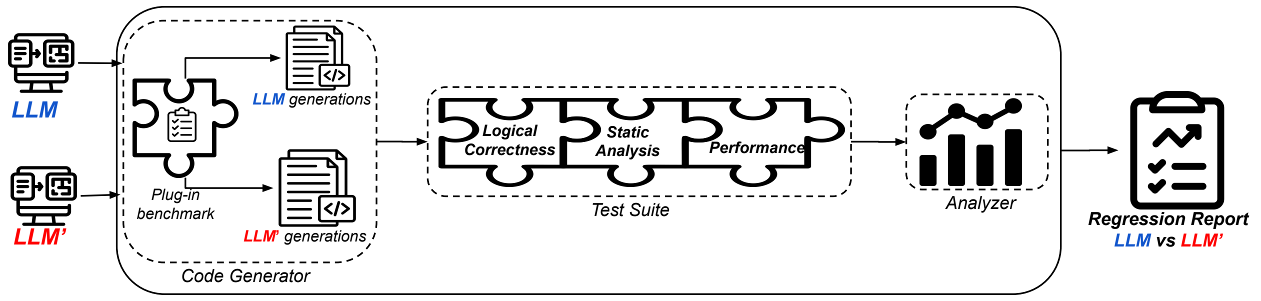


Figure 6.1 ReCatcher Architecture: LLM Regression Testing Framework for Code Generation

Code Generator

This component is responsible for generating code using the two LLMs under test (LLM and LLM') based on a specified evaluation benchmark consisting of programming tasks. Since LLM-generated code can be non-deterministic, the generation process is repeated `m` times to mitigate randomness and ensure result consistency. The default repetition number is 10, however, users can adjust according to their available resources and evaluation needs.

ReCatcher allows users to integrate different code generation benchmarks. To ensure compatibility, the evaluation benchmark must meet two conditions: (i) be specifically designed for Python code generation tasks, as the framework currently focuses exclusively on Python; and (ii) include unit tests for each task, which are crucial for verifying logical correctness and assessing performance. To extend ReCatcher with a new benchmark, users must implement an interface that allows the framework to interact with the benchmark and extract unit tests. This integration ensures that ReCatcher can evaluate code generation against the newly added benchmark. ReCatcher currently supports HumanEval+ [37] and BigCodeBench [131], but additional benchmarks can be integrated. By supporting diverse evaluation scenarios, ReCatcher ensures meaningful, actionable results across various use cases, making it a flexible and extensible regression testing framework for LLM-generated code.

Test Suite

This component is the core of ReCatcher, responsible for assessing LLMs' code generation capability using the tests defined in Subsection 6.1. The *Test Suite* runs the tests on the code generated by *Code Generator* across three code aspects: logical correctness, static code issues, and performance. The collected test outputs are then passed to the component, *Analyzer*. For logical correctness and static code issues, each test is conducted independently on every generated code snippet for each LLM, task, and repetition.

Although unit tests, which assess logical correctness, can sometimes exhibit flaky behavior, we did not observe flakiness in our experiments. Therefore, each unit test is executed once per generated code snippet. As output, *Test Suite* outputs a labeled dataset, where each snippet is annotated with detected inefficiencies. These results are aggregated to track inefficiency occurrences across repetitions and LLMs for further analysis by the *Analyzer* component.

For performance testing, only functionally correct code is considered. If a generated code is incorrect, its execution time or memory usage is irrelevant, as it does not fulfill the intended task and could introduce misleading conclusions; for example, an LLM generating a faster yet incorrect solution would not be a meaningful improvement. To ensure fair and reliable comparisons, performance tests are conducted only when both LLMs generate correct code for a given benchmark task. To mitigate execution randomness, each test is repeated `n` times. The default test repetition is 5, however users can adjust according to their available resources. Performance evaluation, supports two types of input data. If available, users can provide large inputs in JSONL format to test LLMs' performance under more demanding conditions. The *Test Suite* executes the generated codes with large inputs, enabling performance assessment under more demanding conditions. If large inputs are not available, *Test Suite* utilizes all inputs from existing unit tests to conduct performance testing. In our study, large inputs were not available, so we conducted performance testing using inputs from unit tests provided with the code generation benchmark. *Test Suite* generates a labeled dataset associating each task with correct outputs from both LLMs with the distribution of execution time and memory usage per LLM.

Test Suite is highly customizable to accommodate different user needs: tests can be enabled or disabled based on the user's evaluation priorities or resource constraints. Additionally, *Test Suite* is easily extensible, allowing users to integrate custom tests by providing a simple interface, ensuring smooth and consistent functionality within the ReCatcher framework.

Analyzer

This component processes the detailed outputs from *Test Suite* component and generates a comprehensive report comparing the code generation capability of the two models under evaluation (LLM and LLM').

For logical correctness and static code issues, the *Analyzer* takes as input the number of inefficiencies detected per repetition for each model.

The analysis begins by calculating the average number of detected inefficiencies per LLM, then quantifying the impact using the percentage:

$$\text{Percentage Difference} = \frac{\overline{\text{Inefficiencies}}_{\text{LLM}} - \overline{\text{Inefficiencies}}_{\text{LLM'}}}{\text{Total Tasks in Benchmark}} \times 100$$

A positive value indicates that LLM' improves over LLM, whereas a negative value suggests that LLM' introduces regressions. The percentage differences are calculated per tested code efficiency, providing fine-grained insights into areas where improvements or regressions occur. For performance testing, the *Analyzer* evaluates the execution time and memory usage distributions for each task to determine whether one LLM-generated code significantly outperforms the other. The task performance distribution is obtained from the *Test Suite* component.

Since we consider only functionally correct code snippets for a given task, the resulting samples are independent non-paired. Under this condition, we apply Mann-Whitney U test, a non-parametric statistical method designed for comparing two independent distributions. Unlike parametric tests, Mann-Whitney U does not assume normality, making it more appropriate for assessing differences in execution time and memory usage distributions between LLM-generated code. The Mann-Whitney U test evaluates the statistical significance of performance differences, determining whether an observed change indicates an improvement, regression, or no significant difference. To ensure meaningful comparisons, we compute the ratio of tasks where LLM' exhibits improvements or regressions by normalizing against the

total number of tasks in the benchmark. This normalization ensures that results remain interpretable across different benchmarks and task distributions.
ReCatcher's final report includes two key metrics: (i) the inefficiency difference rate of inefficiency differences observed for logical correctness and static code analysis between LLM and LLM' per test; and (ii) the ratio of tasks exhibiting performance regressions or improvements based on execution time and memory usage. For users requiring more detailed and advanced analysis beyond aggregated metrics, the *Test Suite* logs allow deeper investigation.

6.3 Experimental Setup and Results

In this section, we present how we use ReCatcher (see Subsection 6.2) to assess regression in LLMs-based code generation. We evaluate three update scenarios for models:

- **Scenario 1: Fine-tuning Impact** – Examining how fine-tuning affects code generation capabilities of LLMs.
- **Scenario 2: Merging Impact** – Examining how merging affects LLMs-based code generation.
- **Scenario 3: Model Release Impact** – Evaluating regressions introduced by new versions of a model within a family of LLMs.

Initially, we detail the model and benchmark selection, then the experiments setup and potential regression in code generation capability.

6.3.1 Model Selection

In this work, we study regression across three update scenarios for models: fine-tuning, merging, and family model releases, for both open-source and proprietary models. For open-source models, we selected CodeLlama-7B and DeepSeek-Coder-6.7B as base models due to their widespread adoption and strong code generation capability on leaderboards among 7B-parameter models [93]. To effectively assess the influence of each scenario, we examined them separately. Specifically, to isolate the impact of fine-tuning and merging, we choose two variants per model (one fine-tuned and the other merged) from the HuggingFace Hub [237]. We prioritized variants that resulted from a single adaptation technique (either fine-tuning or merging) and we reviewed their details (e.g., fine-tuning datasets, and merging methods) through model cards. Selection was based on the highest number of downloads in Nov 2025 (last checked on Nov 27, 2024), with likes as a tiebreaker. The fine-tuned variants were adapted using the Kotlin Exercises dataset [264], while the merged variants included: (i)

CodeLlama-7B merged with Llama2-7B, and (ii) DeepSeek-Coder-7B merged with DeepSeek-Coder Instruct and OpenCodeInterpreter-7B, both merged using the TIES technique [99]. For proprietary models, we selected GPT family models: GPT-3.5-turbo, GPT-4o, and GPT-4o-mini. When conducting this study, GPT-3.5 and GPT-4 represent the latest major releases, while GPT-4o-mini was included for its cost-efficiency and comparable performance to GPT-4o [93].

6.3.2 Benchmark Selection

We selected HumanEval+ [37] and BigCodeBench [131] as benchmarks to configure the Code Generator component in *ReCatcher*. HumanEval+ is a widely used benchmark consisting of 164 manually crafted algorithmic tasks, each with unit tests for correctness assessment. While effective in evaluating code generation [23], HumanEval+ is limited to algorithmic tasks and does not include real-world software development challenges. To address this limitation, we included BigCodeBench [131], which features 1,140 fine-grained tasks. The tasks require LLMs to invoke functions from 139 libraries across 7 different domains. Each task includes test cases, making it well-suited for effective evaluation of code generation. By combining HumanEval+ and BigCodeBench, we ensure a robust evaluation across both algorithmic and real-world programming tasks.

6.3.3 Experimental Setup

To systematically evaluate LLM capabilities through code inefficiencies, we designed a structured experimental setup covering code generation, test execution, and hardware configurations.

Code-Generation Setup

When using the models to generate code for the benchmarks' tasks, we consider the following configurations:

- **temperature = 0.1**, to provide more accurate and deterministic outputs with minimal randomness [199];
- **max tokens = 2048**, aiming to accommodate sufficiently long code generations;
- **top-p = 0.95**, to balance diversity and coherence in generated outputs;
- **default prompt**, the method signature and task description from the benchmarks serve as input for the LLMs;
- **model access**, GPT models were queried on January 18, 2025.

ReCatcher Setup

Aiming to support the reproducibility of our results and accounting for variations in generation and execution, we applied multiple trials:

- **Code Generation Repetitions (n=10)**, each prompt (task) is generated 10 times per model;
- **Performance Test Repetition (m=5)**, each generated code snippet is executed 5 times.

Hardware Setup

To run our experiments, while aiming to reduce costs, we adopted different strategies, specifically for the tasks in BigCodeBench, as they require more powerful computational resources. Below, we detail the hardware setup in our study:

- **Code Generation:** NVIDIA A100 (32GB);
- **Static Tests:** 12 vCPUs with 32GB RAM;
- **General Logic and Performance Tests:** NVIDIA V100 (32GB).

6.3.4 Results

Table 6.1 summarizes the regression test results across the three update scenarios: fine-tuning, merging, and model release. In the following, we analyze the results for each scenario, highlighting key trends, regressions, and improvements in code generation capability.

Scenario 1: Fine-tuning Impact

For evaluating the impact of fine-tuning, we used (i) a fine-tuned variant of CodeLlama and (ii) a fine-tuned variant of DeepSeek-Coder, both fine-tuned on the same dataset, Kotlin Exercises [264]. The most significant regression between the original and fine-tuned variant is observed in *Syntax Error*. For example, the fine-tuned variant of CodeLlama, when tested on HumanEval+, showed 12.93% regression in *Syntax Error* indicating a declined capability to generate syntactically correct Python code. This regression, although varying in intensity, is present across the two used benchmarks and also on the fine-tuned variant of DeepSeek-Coder. A possible explanation is the nature of the fine-tuning dataset, which consists of Kotlin code, whereas the evaluation is performed on Python code. Given the significant syntactic differences between Kotlin and Python, fine-tuning may have disrupted the model's learned syntax, leading to more frequent syntax errors. However, for *Missing*

Table 6.1 Regression Testing Results Across Model Update Scenarios

			CodeLlama		DeepSeek-Coder		GPT Family	
			Fintuned vs Original	Merged vs Original	Fintuned vs Original	Merged vs Original	GPT 3.5 vs 4o	GPT 4o vs 4o-mini
HumanEval+	General Logic	Incorrect Code	1.95%	-18.72%	14.94%	25.98%	4.15%	3.48%
	Errros	Syntax Error	-12.93%	-9.02%	-1.04%	1.22%	-0.12%	0.55%
		Missing Declaration/Import	-1.89%	3.54%	1.16%	0.79%	-10.73%	10.55%
	Maintainability	Code Duplication	2.07%	-2.87%	1.16%	1.65%	5.06%	-0.61%
		Comment Duplication	4.51%	-4.45%	2.44%	2.50%	0.00%	0.00%
		Unnecessary Else	16.22%	7.80%	7.87%	7.44%	4.39%	-0.24%
		Unnecessary Coditional Block	0.98%	1.89%	0.67%	1.10%	0.85%	0.18%
	Readability	Confusing Variable Naming	1.22%	-0.18%	0.55%	-1.28%	2.87%	0.00%
		Sub-readable Code	1.59%	1.22%	2.93%	2.38%	0.67%	0.00%
	Performance	Memory Usage: Improvement	0.61%	0.00%	0.61%	0.00%	0.61%	0.00%
		Memory Usage: Regression	0.00%	0.00%	1.83%	1.22%	0.00%	0.61%
		Execution Time Improvement	26.22%	10.98%	46.34%	48.17%	65.24%	0.61%
		Execution Time Regression	0.61%	0.00%	0.61%	1.22%	0.00%	80.49%
BigCodeBench	General Logic	Incorrect Code	3.41%	6.71%	9.32%	1.95%	-12.19%	11.39%
	Errros	Syntax Error	-0.69%	-30.77%	-3.44%	0.26%	-2.62%	-0.25%
		Missing Declaration/Import	3.38%	2.55%	1.65%	1.54%	-51.64%	28.31%
	Maintainability	Code Duplication	3.47%	-20.36%	1.49%	0.72%	1.13%	-1.18%
		Comment Duplication	10.16%	1.39%	0.39%	0.40%	0.03%	0.00%
		Unnecessary Else	1.90%	-1.90%	0.10%	0.08%	1.07%	0.78%
		Unnecessary Conditional Block	-0.69%	-0.21%	-0.59%	0.01%	0.15%	-0.01%
	Readability	Confusing Variable Naming	-1.24%	3.60%	0.40%	0.31%	0.88%	-0.23%
		Sub-readable Code	-0.07%	-0.14%	0.44%	0.30%	0.57%	-0.02%
	Performance	Memory Usage: Improvement	0.35%	2.28%	1.23%	2.72%	0.96%	0.61%
		Memory Usage: Regression	0.61%	1.49%	0.61%	9.04%	0.88%	1.32%
		Execution Time Improvement	9.74%	26.84%	32.54%	34.82%	10.44%	21.14%
		Execution Time Regression	11.93%	1.32%	8.16%	5.53%	20.00%	15.35%

Declaration/Import, although related to programming language, we did not notice regression (only fine-tuned CodeLlama on HumanEval+, which showed only 1.89% regression), instead, we noticed a minor improvement in handling missing imports with a maximum of 3.38%. These results suggest that while *Syntax Error* were heavily impacted by language differences, *Missing Declaration/Import* remained relatively stable, likely because such errors are linked to semantic understanding rather than strict syntax rules.
Despite the regression in *Syntax Error*, we observed an overall improvement in *General Logic*. The improvement regarding *General Logic* may be greater than the reported values since some logically correct implementations could not be evaluated due to syntax-related execution failures. Regarding *Maintainability*, fine-tuned variants showed improvement particularly in *Comment Duplication*. The CodeLlama variant improved by 10.16% in *Comment Duplication* on BigCodeBench, while DeepSeek-Coder showed an improvement, though to a lesser extent. One possible reason for this improvement is that DeepSeek-Coder was trained from scratch on a large-scale code dataset, making it better at handling redundancy and duplication. Additionally, we observed an improvement in *Unnecessary Else* statements, by 16.22% and 7.87%, for the fine-tuned variant of CodeLlama and DeepSeek-Coder, respectively. Such an improvement suggests that fine-tuning, even on a different programming language, may help LLMs generate more concise and structured conditional statements, improving maintainability.
The impact of fine-tuning on *Readability* was limited, with percentage difference ranging between -1.24% and 2.93% across models. In terms of memory usage, we observed no significant difference across models. However, for execution time, fine-tuning had mixed effects. The DeepSeek-Coder fine-tuned model on BigCodeBench showed a 46.34% improvement in execution time, whereas the fine-tuned CodeLlama model exhibited a slight 2% net regression (+9.74% and -11.93%, for improvement and regression, respectively). Our results suggest that the impact of fine-tuning remains consistent when the same dataset is used for fine-tuning. However, when fine-tuning is performed on coding tasks from different programming languages, it can enhance logical reasoning but may also introduce syntax errors.

Scenario 2: Merging Impact

For evaluating the merging adaptation, we considered two merged variants: (i) CodeLlama merged with Llama2 and (ii) DeepSeek-Coder merged with OpenCodeInterpreter [204].
The CodeLlama merged variant exhibited a regression of 30.77% in *Syntax Error* on BigCodeBench and 18.72% on HumanEval+. This regression may be due to the merging process incorporating a general-purpose model (Llama2) not optimized for code generation, leading to syntax inconsistencies. The regression on *Syntax Error* may explain the regression of 18.72%

in logical correctness on HumanEval+. However *General Logic* improved by 6.71% on BigCodeBench. The actual improvement might be higher but we cannot precisely quantify it, because of syntax errors. The merging impact difference between benchmarks may indicate that *General Logic* is more affected in algorithmic-style tasks (as seen in HumanEval+) rather than in real-world programming tasks spanning diverse domains such as machine learning and data analytics (as covered by BigCodeBench). Regarding *Missing Declaration/Import*, we observe an improvement of 3.54% when tested on HumanEval+, suggesting that merging impact on *Syntax Error* is more pronounced than on *Missing Declaration/Import*.
For *Maintainability*, we observed a significant regression of 20.36% in *Code Duplication* on BigCodeBench. This regression suggests that merging with general-purpose LLM, which lacks code-specific optimization, may lead to inconsistent coding patterns, increasing redundancy. Regarding *Readability*, the impact remained minimal with differences rates ranging from 3.60% to -0.18%, highlighting that merging influences the syntax structure more than the overall readability. For *Performance*, memory usage remained unchanged, while execution time improved by 26.84% on BigCodeBench and 10.98% on HumanEval+.
Regarding the DeepSeek-Coder merged model, we observed overall improvements across all code aspects except for a minor minor exception regression regression of 1.28% in *Confusing Variable Naming* on HumanEval+. The improvements were overall more pronounced on HumanEval+, while BigCodeBench shows overall marginal improvements, possibly because the merged model (OpenCodeInterpreter) was trained on LeetCode-style programmatic tasks [204], which resembles HumanEval+. *General Logic* showed improvement of 25.98% on HumanEval+ and only 2% on BigCodeBench. *Performance* showed a notable improvement of 48.18% on HumanEval+ and 34.82% on BigCodeBench regarding execution time. This improvement may suggest that merging with a model trained on LeetCode-style tasks emphasizes performance optimization, as these tasks prioritize algorithmic efficiency, minimal computation, and optimized execution paths. However, memory usage showed a net regression of approximately 6% on BigCodeBench and only 1% HumanEval+. For *Unnecessary Else*, we observed an improvement of 7.87% when tested on HumanEval+, while results remain stable on BigCodeBench. For the other code aspects, *Readability*, *Maintainability* and *Errors*, we have seen minimal variations with differences in rates lower than 2%.
These findings emphasize that model selection plays a crucial role in merging, as it can either mitigate or amplify regressions in code generation capabilities.

Scenario 3: Model Release Impact

To assess the impact of model release, we studied two release scenarios: (i) migration from GPT-3.5-turbo to GPT-4o and (ii) migration from GPT-4o to GPT-4o-mini.

Experiment 1: Migration from GPT-3.5-turbo to GPT-4o GPT-4o exhibited a significant regression in *Missing Declaration/Import* of 10.73% on HumanEval+ and 51.64% on BigCodeBench. The regression is more pronounced in BigCodeBench, which can be attributed to its greater diversity across tasks beyond algorithmic problem-solving, requiring a wider range of libraries. Prior studies have also highlighted GPT models' challenges in handling external libraries and issues with deprecated or incorrectly referenced methods, further reinforcing this observation [219]. We also observed a small regression in *Syntax Errors* of 2.62% on BigCodeBench. Both *Missing Declaration/Import* and *Syntax Error* directly impact the assessment of *General Logic* by preventing the execution of test cases, as these errors cause the program to fail during runtime. The combined effect of these regressions in *Missing Declaration/Import* and *Syntax Error* contributed to the 12.19% regression in *General Logic* on BigCodeBench. Interestingly, although HumanEval+ exhibited a 10.73% regression in *Missing Declaration/Import*, it still showed a 4.15% improvement in *General Logic*. Regarding *Performance*, we observed a notable improvement in execution time efficiency by 65.24% on HumanEval+ and a net regression of approximately 10% on BigCodeBench. Finally, GPT-4o demonstrated a slight improvement in *Maintainability*, of 5.06% in *Code Duplication*, suggesting a minor structural improvement in generated code despite other regressions. Although GPT-4o exhibited regressions across multiple aspects, it is still widely considered more capable than GPT-3.5-turbo. These findings highlight the importance of a comprehensive regression testing framework that evaluates models beyond predefined correctness benchmarks, ensuring a well-rounded assessment of code quality and performance.

Experiment 2: Migration from GPT-4o to GPT-4o-mini GPT-4o-mini demonstrated minimal variations when compared with GPT-4o. *Readability*, *Maintainability*, and *Syntax Error* remained largely unchanged, indicating that the smaller model preserved many of GPT-4o's code generation capabilities. Despite being a smaller and more cost-effective, GPT-4o-mini outperforms GPT-4o showing an 11.39% improvement in *General Logic* on BigCodeBench. Additionally, its handling of *Missing Declaration/Import* improved by up to 28.31% on BigCodeBench. However, the most significant regression was observed in execution time. GPT-4o-mini exhibited an 80.49% regression on HumanEval+, while showing a slight improvement of 5% on BigCodeBench. This regression indicates that the execution time of code snippets generated by GPT-4o-mini increased significantly for algorithmic tasks that require optimization, such as those in HumanEval+.

6.4 Key Takeaways

This study highlights that while LLMs' code generation capability is continuously evolving, improvements and regressions occur across various code aspects. Trade-offs are common, where gains in one aspect may come at the expense of regressions in another. The most volatile code aspects we observed are *General Logic*, *Errors* (*Syntax Error*, *Missing Declaration/Import*) and *Execution Time*. Based on our findings, we recommend prioritizing these aspects in regression testing before migration. However, memory usage remained relatively stable across most model updates. Comprehensive testing is essential before adopting a new model, as higher capability claims do not always translate to improvements in all code aspects. For instance, GPT-4o exhibited regressions compared to GPT-3.5-turbo, underscoring the need for empirical validation over assumed superiority.
Regarding the model adaptations, for fine-tuning, selecting a training dataset that aligns with the target programming language is crucial. Mismatches between the fine-tuning dataset and the intended use case can introduce syntax errors and logic inconsistencies. For model merging, reviewing the training process of the merged models is essential, as the choice of merged models directly impacts code generation capability. To preserve code quality and minimize regressions, merging should prioritize models trained specifically for coding tasks, ideally within a domain relevant to the user's needs.
Regarding the applicability of *ReCatcher*, we advocate that it provides a structured framework for practitioners and model maintainers to assess trade-offs introduced by model updates. By systematically analyzing regressions across multiple code aspects, ReCatcher can serve as a benchmarking tool or integrated into a leaderboard enabling continuous tracking of model performance over time.

6.5 Threats to Validity

Construct Validity. The proposed test for detecting static code inefficiencies may introduce faults, potentially affecting the results. To address this, we relied on proved and widely used tools: Pylint [235] for static analysis and CPD [252] for duplication detection which has been validated in previous research [255–258]. For performance testing, we acknowledge that executing unit tests alone does not fully capture performance bottlenecks. To address this, *ReCatcher* also supports testing using large inputs. For logical correctness, we recognize that unit tests may have limited coverage. However, this can be improved by generating additional tests using tools such as EvoSuite [233]. Finally, *ReCatcher* builds its tests on a taxonomy of inefficiencies, which covers a broad range of issues but may still overlook certain aspects. To address this, we designed *ReCatcher* to be easily extensible, allowing users to integrate additional tests as needed.

Internal Validity. Non-determinism in LLM-generated outputs may introduce bias in our results. To mitigate this, we repeated the generation process 10 times. Additionally, we set the temperature to 0.1 to limit randomness and reduce variability in outputs. For performance evaluation, each execution was repeated 5 times to minimize fluctuations. To ensure that performance measurements were not affected by external system interference, we conducted the assessments in an isolated environment with no other running processes. Instead of relying solely on average comparisons, we used *Mann-Whitney U test* to ensure the reliability of our findings. This approach provides a more robust assessment of regression between models, accounting for variations beyond simple mean comparisons. Another potential threat is the interdependence of test results. Certain inefficiencies, such as syntax errors and missing declarations/imports, directly impact logical correctness. This interdependency could misestimate logical correctness issues, as some logically correct code may not reach execution. Automatically assessing logical correctness in the presence of syntax errors and missing declarations is inherently challenging. However, if a code snippet requires manual intervention for basic fixes before execution, it is not fully reliable, making its logical correctness evaluation less relevant in practical scenarios.

External Validity. Our study focuses on a specific set of models, analyzing their behavior under fine-tuning, merging, and new model releases. While these insights are valuable for understanding model regression, our findings may not generalize to LLMs with different architectures, training strategies, or fine-tuning datasets. To evaluate fine-tuning and merging, we selected one fine-tuned and one merged variant per model. Selection was based on the highest number of downloads and well-documented model cards available on Hugging Face. While this ensures that the chosen models are widely used and sufficiently documented, other fine-tuned or merged variants may exhibit different behaviors. Our findings are also influenced by the chosen benchmarks, which may limit generalizability. To mitigate this, we used two complementary benchmarks: HumanEval+ [37], a widely adopted benchmark for assessing LLM code generation performance, and BigCodeBench [131], which includes a diverse set of real-world programming tasks beyond algorithmic problem-solving. To enhance reproducibility and support research, we provide a replication package [238] containing the generated code snippets, test results, and the *ReCatcher* source code.

Conclusion Validity. Our conclusions are based on quantitative analysis using automated tools, relying on Hugging Face model cards for information about model fine-tuning and merging. However, since not all model cards provide complete details, this could influence the interpretation of certain results. To mitigate this, we selected only highly downloaded Hugging Face models with comprehensive model cards that included details on fine-tuning datasets, merged models, and applied techniques. For OpenAI models, we conducted a thorough review of their website and research papers to gather the necessary information.

6.6 Chapter Summary

In this chapter, we introduced *ReCatcher*, the first systematic LLMs regression testing framework for python code generation. *ReCatcher* enables a comprehensive regression testing between LLMs by assessing logical correctness, static code quality, and performance efficiency. We used *ReCatcher* to test regression across three scenarios: fine-tuning, merging, and model release on CodeLlama, DeepSeek-Coder and GPT-family models. Our findings highlight key trends: (i) the dataset used for fine-tuning can introduce syntax regression, (ii) model merging with general purpose LLM may amplify overall regression, and (iii) GPT-4o exhibits regressions in handling missing imports compared to GPT-3.5-turbo, while GPT-4o-mini shows performance regressions in execution time. By detecting regressions, *ReCatcher* enables practitioners to detect reliability risks before model deployment—making LLMOps pipelines more robust and trustworthy. It allows developers and teams to track regressions beyond correctness and empowers them to make informed decisions when updating or integrating LLMs in real-world systems.
ReCatcher is open-sourced to encourage adoption and further research, and we envision future enhancements including distribution-aware regression detection and task-difficulty–sensitive analysis. These directions aim to provide even deeper insights into LLM evolution and strengthen the reliability of LLMOps pipelines for code generation tasks.

CHAPTER 7 CONCLUSION

In this chapter, we conclude the thesis and summarize our findings. In addition, we will discuss the limitations of our studies and the directions for future work

7.1 Summary

ML models and LLMs have demonstrated their ability to automate a wide range of tasks, from industrial inspection to code generation. With their growing capabilities, these models are increasingly integrated into larger AI-based systems. To ensure reliable and trustworthy systems in dynamic production environments, it is essential that the underlying models are continuously delivered and updated with more capable or better-adapted models to the production environment. Managing the lifecycle of ML models and LLMs has given rise to the fields of MLOps and LLMOps, which provide a set of tools and best practices to automate various stages of model development, deployment, and monitoring. These practices include CT to adapt to evolving data, automated models evaluation on production data to ensure that newly deployed models do not degrade in performance. While MLOps and LLMOps offer significant benefits in terms of automation and scalability, they also introduce limitations that can compromise the reliability of AI-based systems if not carefully managed. For instance, incorporating noisy, drifted, or incorrectly auto-labeled data into a CT workflow can lead to catastrophic forgetting. Inadequate validation of newly trained models before deployment may result in misleading evaluations and ultimately lead to performance degradation.
This thesis aims to enhance the reliability and trustworthiness of MLOps and LLMOps pipelines. More specifically, it introduces: (i) *MLOps* - a CT-based MLOps pipeline maintenance approach, which leverages a two-stage filtering mechanism to automatically label data and eliminate drifted or noisy inputs, ensuring stable model updates in dynamic environments; and (ii) *LLMOps* - a comprehensive taxonomy of inefficiencies in LLM-generated code, along with a regression testing framework that enables more reliable model evaluation and safer model updates, ultimately supporting dependable LLMOps workflows
In the following we provide an overview of the studies and approaches presented in the thesis.

- *Trimming the Risk: Towards Reliable Continuous Training for Deep Learning Inspection Systems*

  We introduce a novel two-stage filter to trim the risks in CT using self-generated labels in dynamic production environment. First, we use calibrated confidence scores to eliminate risky data. Second, we employ Variational Auto Encoder (VAE) and Histogram to encode rich image embeddings to filter significantly shifted risky data. The accepted subset is suitable for reliable maintenance of inspection system models. Our evaluation on two industrial inspection systems demonstrates that our approach passes less than 9% of false predictions, leading to an increase of F1-score up to 14% on critical production data. We discover limitations of supervised learning algorithms such as their inability to self-learn new defects and their reliance on human verification. Even if our reliable CT approach is applied for successive cycles, the model could not capture new defect patterns that are not similar to the ones from in-distribution.

- *A Taxonomy of Inefficiencies in LLM-Generated Code*

  We systematically investigated inefficiencies in LLM-generated code and introduced a taxonomy categorizing them into 5 categories: *General Logic*, *Performance*, *Readability*, *Maintainability*, and *Errors*, with 19 specific subcategories. We validated the taxonomy through a survey of 58 practitioners and researchers, who largely confirmed its completeness and affirmed the relevance and prevalence of the identified inefficiencies in real-world scenarios. Our findings reveal a critical gap in the ability of LLMs to generate efficient code. *General Logic* and *Performance* inefficiencies emerged as the most frequent and relevant inefficiencies, often co-occurring with *Maintainability* and *Readability* highlighting their impact on overall LLM-generated code quality. This study provides a structured basis for evaluating LLM-generated code, facilitating efficiency comparisons, and guiding future improvements in LLMs for code generation.

- *ReCatcher: Towards LLMs Regression Testing for Code Generation*

  We propose *ReCatcher*, the first systematic LLMs regression testing framework for python code generation. *ReCatcher* enables a comprehensive regression testing between LLMs by assessing logical correctness, static code quality, and performance efficiency. We used *ReCatcher* to test regression across three scenarios: fine-tuning, merging, and model release on CodeLlama, DeepSeek-Coder and GPT-family models. Our findings highlight key trends: (i) the dataset used for fine-tuning can introduce syntax regression, (ii) model merging with general purpose LLM may amplify overall regression, and (iii) GPT-4o exhibits regressions in handling missing imports compared to GPT-3.5-turbo, while GPT-4o-mini shows performance regressions in execution time. We believe *ReCatcher* can promote further research and real-world adoption, assisting users in making informed decisions about model updates.

The research studies presented in this thesis jointly work toward strengthening the reliability and trustworthiness of AI-based systems by addressing key gaps in both MLOps and LLMOps practices. In the context of MLOps, our proposed CT-based maintenance approach mitigates the risks of performance degradation due to noisy or shifted data in dynamic environments. By introducing a robust two-stage filtering mechanism, this work ensures that only reliable and relevant samples are used for model updates—thereby supporting safer and more stable deployment cycles in industrial systems. On the LLMOps side, the taxonomy of inefficiencies in LLM-generated code provides a foundational framework for understanding and categorizing reliability concerns in code outputs. It offers actionable insights for both researchers and practitioners to evaluate LLM performance beyond accuracy. Furthermore, the ReCatcher framework introduces the much-needed ability to perform systematic regression testing across LLM variants and versions. This allows for safer model updates by detecting regressions in logic, performance, or code quality—paving the way for more robust LLMOps pipelines. Together, these contributions represent meaningful steps toward building dependable pipelines for developing, deploying, and evolving ML and LLM models. They advocate for structured evaluation, continuous monitoring, and safe adaptation—cornerstones for trustworthy and reliable AI-based systems.

7.2 Limitations and Future Work

In this section, we discuss the limitations of our studies and potential directions for future research.

Limitations

In the MLOps context, our CT pipeline was evaluated exclusively on visual inspection systems. While these systems are representative of real-world industrial deployments, the generalizability of the approach to other domains remains untested. Furthermore, although our two-stage filtering approach was shown to effectively reduce the risks of performance degradation by eliminating noisy or drifted data, we did not benchmark it against existing drift mitigation strategies. This limits our ability to quantify the exact improvements introduced by our pipeline. Another limitation is the manual calibration of confidence and distribution thresholds used during filtering. These thresholds were tuned empirically for our datasets, but such manual processes may not scale across different environments or adapt dynamically as data evolves.

In the LLMOps part of this thesis, the inefficiency taxonomy and ReCatcher framework were developed and validated using only Python code. While Python is widely used, the conclusions may not hold for other languages or domains. Additionally, the regression testing framework was evaluated in isolation from real-world software development workflows. We did not assess how developers interact with the tool in iterative code generation settings or how it integrates into their day-to-day LLMOps pipelines. As such, the usability and adoption potential of ReCatcher in practice remains to be validated.

Future Work

There are several promising directions for future research building on this work. In the context of MLOps, our continuous training (CT) pipeline could be extended to additional domains such as medical imaging and remote sensing, which are also affected by distribution shifts and evolving data conditions. Evaluating the approach in these settings would offer deeper insights into its robustness, adaptability, and generalizability beyond industrial inspection. To broaden the applicability of the pipeline, future work may explore its adaptation to unsupervised learning models, which are increasingly adopted in visual inspection and other domains. Supporting unsupervised settings would require rethinking both drift detection and data selection strategies in the absence of labels. The current pipeline's reliance on manually calibrated thresholds presents a scalability challenge. Integrating adaptive or data-driven thresholding techniques—such as Bayesian optimization, reinforcement learning, or meta-learning—could improve automation by dynamically adjusting thresholds based on observed drift severity or model feedback. Further improvements may stem from investigating advanced auto-labeling strategies. Rather than relying solely on confidence calibration, future approaches could integrate drift detection, mislabeled sample identification, and self-supervised or semi-supervised labeling to enhance labeling accuracy and coverage. Integrating online or continual learning algorithms also represents a key opportunity. These would enable the system to incrementally adapt to incoming data streams, reducing dependence on periodic retraining and improving responsiveness in production environments. To optimize training efficiency, smarter data selection methods could be adopted. Techniques like data distillation, core-set selection, or importance-based sampling can help prioritize the most informative samples, avoiding redundant or low-impact retraining. Finally, extending the pipeline with rigorous model validation under real-world deployment constraints—such as A/B testing, canary releases, or shadow deployments—can provide controlled, empirical comparisons between retrained and baseline models. These strategies help ensure performance and reliability prior to full-scale rollout.

Regarding LLMOps, several research avenues emerge from our findings. One direction involves expanding the inefficiency taxonomy to cover a broader set of programming languages and code generation tasks, including those related to web development, machine learning pipelines, and security-sensitive applications. This expansion would improve the taxonomy's generalizability and practical value. Another opportunity lies in comparing inefficiencies in human-written versus LLM-generated code, which may uncover whether LLMs intro-

duce novel inefficiencies or merely reproduce patterns found in human code. Our ReCatcher framework could also be extended to support more programming languages and incorporate evaluations of non-functional aspects such as readability, maintainability, and performance. Enhancing the tool with a graphical user interface could further facilitate adoption by developers. To encourage broader model assessment, a public leaderboard could be introduced to benchmark LLMs across a range of inefficiency metrics—beyond correctness—enabling more informed model selection by practitioners and researchers. Deeper investigation into the root causes of regressions in LLM variants, including the roles of training data, fine-tuning strategies, and architectural changes, could help illuminate model reliability. Integrating regression testing into development workflows and incorporating user feedback would also provide practical insights into tool usability and effectiveness.
Finally, expanding the reliability evaluation scope to include additional dimensions—such as fairness, explainability, or adversarial robustness—would support a more holistic understanding of trustworthiness in MLOps and LLMOps pipelines. Combining our CT strategy with complementary retraining methods or alternative drift detection techniques may further lead to resilient hybrid systems.

REFERENCES

[1] D. Kedziora and S. Hyrynsalmi, “Turning robotic process automation onto intelligent automation with machine learning,” in *Proceedings of the 11th International Conference on Communities and Technologies*, 2023, pp. 1–5.

[2] Y. Liu, C. Zhang, and X. Dong, “A survey of real-time surface defect inspection methods based on deep learning,” *Artificial Intelligence Review*, vol. 56, no. 10, pp. 12 131–12 170, 2023.

[3] F. Lupi, M. Biancalana, A. Rossi, and M. Lanzetta, “A framework for flexible and reconfigurable vision inspection systems,” *The International Journal of Advanced Manufacturing Technology*, vol. 129, no. 1, pp. 871–897, 2023.

[4] A. Bandi, P. V. S. R. Adapa, and Y. E. V. P. K. Kuchi, “The power of generative ai: A review of requirements, models, input–output formats, evaluation metrics, and challenges,” *Future Internet*, vol. 15, no. 8, 2023.

[5] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, S. Zhong, B. Yin, and X. Hu, “Harnessing the power of llms in practice: A survey on chatgpt and beyond,” *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 6, pp. 1–32, 2024.

[6] H. Thadeshwar, V. Shah, M. Jain, R. Chaudhari, and V. Badgujar, “Artificial intelligence based self-driving car,” in *2020 4th international conference on computer, communication and signal processing (ICCCSP)*. IEEE, 2020, pp. 1–5.

[7] J. Lim, H. Lee, Y. Won, and H. Yeon, “{MLOp} lifecycle scheme for vision-based inspection process in manufacturing,” in *2019 USENIX Conference on Operational Machine Learning (OpML 19)*, 2019, pp. 9–11.

[8] G. Prasanna, “Optimizing the future: Unveiling the significance of mlops in streamlining the machine learning lifecycle,” *Int. J. Sci. Res. Eng. Technol*, vol. 4, pp. 5–8, 2024.

[9] S. Alla and S. K. Adari, “What is mlops?” in *Beginning MLOps with MLFlow: Deploy Models in AWS SageMaker, Google Cloud, and Microsoft Azure.* Berkeley, CA: Apress, 2021, pp. 79–124.

[10] D. Kreuzberger, N. Kühl, and S. Hirschl, “Machine learning operations (mlops): Overview, definition, and architecture,” *IEEE access*, vol. 11, pp. 31 866–31 879, 2023.

[11] M. M. John, H. H. Olsson, and J. Bosch, “Towards mlops: A framework and maturity model,” in *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* IEEE, 2021, pp. 1–8.

[12] S. Mäkinen, H. Skogström, E. Laaksonen, and T. Mikkonen, “Who needs mlops: What data scientists seek to accomplish and how can mlops help?” in *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN).* IEEE, 2021, pp. 109–112.

[13] R. Patil and V. Gudivada, “A review of current trends, techniques, and challenges in large language models (llms),” *Applied Sciences*, vol. 14, no. 5, p. 2074, 2024.

[14] V. Komanduri, S. Estropia, S. Alessio, G. Yerdelen, T. Ferreira, G. P. Roldan, Z. Dong, and R. Rojas-Cessa, “Optimizing llm prompts for automation of network management: A user’s perspective,” in *2025 International Conference on Artificial Intelligence in Information and Communication (ICAIIC).* IEEE, 2025, pp. 0958–0963.

[15] F. Ye, M. Yang, J. Pang, L. Wang, D. Wong, E. Yilmaz, S. Shi, and Z. Tu, “Benchmarking llms via uncertainty quantification,” *Advances in Neural Information Processing Systems*, vol. 37, pp. 15 356–15 385, 2024.

[16] J. Stone, R. Patel, F. Ghiasi, S. Mittal, and S. Rahimi, “Navigating mlops: Insights into maturity, lifecycle, tools, and careers,” *arXiv preprint arXiv:2503.15577*, 2025.

[17] G. Araujo, M. Kalinowski, M. Endler, and F. Calefato, “Professional insights into benefits and limitations of implementing mlops principles,” *arXiv preprint arXiv:2403.13115*, 2024.

[18] F. Bayram and B. S. Ahmed, “Towards trustworthy machine learning in production: An overview of the robustness in mlops approach,” *ACM Computing Surveys*, vol. 57, no. 5, pp. 1–35, 2025.

[19] F. Bayram, B. S. Ahmed, and A. Kassler, “From concept drift to model degradation: An overview on performance-aware drift detectors,” *Know.-Based Syst.*, vol. 245, no. C, Jun. 2022.

[20] 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.

[21] W. Ma, C. Yang, and C. Kästner, "(why) is my prompt getting worse? rethinking regression testing for evolving llm apis," in *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*, 2024, pp. 166–171.

[22] R. L. Silva, O. Canciglieri Junior, and M. Rudek, "A road map for planning-deploying machine vision artifacts in the context of industry 4.0," *Journal of Industrial and Production Engineering*, vol. 39, no. 3, pp. 167–180, 2022.

[23] 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.

[24] A. Khadka, S. Sthapit, G. Epiphaniou, and C. Maple, "Resilient machine learning: Advancement, barriers, and opportunities in the nuclear industry," *ACM Computing Surveys*, vol. 56, no. 9, pp. 1–29, 2024.

[25] J. Ding, M. Chen, T. Wang, J. Zhou, X. Fu, and K. Li, "A survey of ai-enabled dynamic manufacturing scheduling: From directed heuristics to autonomous learning," *ACM Comput. Surv.*, vol. 55, no. 14s, Jul. 2023.

[26] S. Shankar, L. Fawaz, K. Gyllstrom, and A. Parameswaran, "Automatic and precise data validation for machine learning," in *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, 2023, pp. 2198–2207.

[27] S. Dong, Q. Wang, S. Sahri, T. Palpanas, and D. Srivastava, "Efficiently mitigating the impact of data drift on machine learning pipelines," *Proceedings of the VLDB Endowment*, vol. 17, no. 11, pp. 3072–3081, 2024.

[28] S. Ackerman, O. Raz, M. Zalmanovici, and A. Zlotnick, "Automatically detecting data drift in machine learning classifiers," *arXiv preprint arXiv:2111.05672*, 2021.

[29] S. Rabanser, S. Günnemann, and Z. Lipton, "Failing loudly: An empirical study of methods for detecting dataset shift," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[30] A. Soin, J. Merkow, J. Long, J. P. Cohen, S. Saligrama, S. Kaiser, S. Borg, I. Tarapov, and M. P. Lungren, "Chexstray: real-time multi-modal data concordance for drift detection in medical imaging ai," *arXiv preprint arXiv:2202.02833*, 2022.

[31] J. Zhang, C.-Y. Hsieh, Y. Yu, C. Zhang, and A. Ratner, "A survey on programmatic weak supervision," *arXiv preprint arXiv:2202.05433*, 2022.

[32] M. Desmond, E. Duesterwald, K. Brimijoin, M. Brachman, and Q. Pan, “Semi-automated data labeling,” in *Proceedings of the NeurIPS 2020 Competition and Demonstration Track*, 2021, pp. 156–169.

[33] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, B. B. Gupta, X. Chen, and X. Wang, “A survey of deep active learning,” *ACM computing surveys (CSUR)*, vol. 54, no. 9, pp. 1–40, 2021.

[34] A. Ratner, S. H. Bach, H. Ehrenberg, J. Fries, S. Wu, and C. Ré, “Snorkel: rapid training data creation with weak supervision,” *The VLDB Journal*, vol. 29, no. 2, pp. 709–730, 2020.

[35] M. Umer, G. Dawson, and R. Polikar, “Targeted forgetting and false memory formation in continual learners through adversarial backdoor attacks,” in *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020, pp. 1–8.

[36] W. Wang, H. Ning, G. Zhang, L. Liu, and Y. Wang, “Rocks coding, not development: A human-centric, experimental evaluation of llm-supported se tasks,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 699–721, 2024.

[37] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[38] 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.

[39] A. Nunez, N. T. Islam, S. K. Jha, and P. Najafirad, “Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing,” *arXiv preprint arXiv:2409.10737*, 2024.

[40] J. Zheng, B. Cao, Z. Ma, R. Pan, H. Lin, Y. Lu, X. Han, and L. Sun, “Beyond correctness: Benchmarking multi-dimensional code generation for large language models,” *arXiv preprint arXiv:2407.11470*, 2024.

[41] S. A. Licorish, A. Bajpai, C. Arora, F. Wang, and K. Tantithamthavorn, “Comparing human and llm generated code: The jury is still out!” *arXiv preprint arXiv:2501.16857*, 2025.

[42] A. Moradi 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.

[43] M. L. Siddiq, L. Roney, J. Zhang, and J. C. D. S. Santos, "Quality assessment of chatgpt generated code and their use by developers," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 152–156.

[44] A. Naik, "On the limitations of embedding based methods for measuring functional correctness for code generation," *arXiv preprint arXiv:2405.01580*, 2024.

[45] J. Diaz-De-Arcaya, J. López-De-Armentia, R. Miñón, I. L. Ojanguren, and A. I. Torre-Bastida, "Large language model operations (llmops): Definition, challenges, and lifecycle management," in *2024 9th International Conference on Smart and Sustainable Technologies (SpliTech)*. IEEE, 2024, pp. 1–4.

[46] Google Cloud, "Mlops: Continuous delivery and automation pipelines in machine learning," Available online, 2025, accessed: 2025-03-24.

[47] S. K. Karmaker, M. M. Hassan, M. J. Smith, L. Xu, C. Zhai, and K. Veeramachaneni, "Automl to date and beyond: Challenges and opportunities," *Acm computing surveys (csur)*, vol. 54, no. 8, pp. 1–36, 2021.

[48] W. C. Potts and C. Carver, "Best practices implementing aiops in large organizations," in *2024 International Conference on Smart Applications, Communications and Networking (SmartNets)*. IEEE, 2024, pp. 1–5.

[49] M. Masana, X. Liu, B. Twardowski, M. Menta, A. D. Bagdanov, and J. Van De Weijer, "Class-incremental learning: survey and performance evaluation on image classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 5, pp. 5513–5533, 2022.

[50] M. Testi, M. Ballabio, E. Frontoni, G. Iannello, S. Moccia, P. Soda, and G. Vessio, "Mlops: A taxonomy and a methodology," *IEEE Access*, vol. 10, pp. 63 606–63 618, 2022.

[51] I. Prapas, B. Derakhshan, A. R. Mahdiraji, and V. Markl, "Continuous training and deployment of deep learning models," *Datenbank-Spektrum*, vol. 21, no. 3, pp. 203–212, 11 2021.

[52] M. Iman, J. A. Miller, K. Rasheed, R. M. Branch, and H. R. Arabnia, "Expanse: A continual and progressive learning system for deep transfer learning," in *2022 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2022, pp. 58–65.

[53] M. Barry, J. Montiel, A. Bifet, N. Manchev, S. Wadkar, M. Halford, R. Chiky, S. Jaouhari, K. B. Shakman, J. Fehaily *et al.*, "Streammlops: Online learning in practice from big data streams & real-time applications," in *39th IEEE International Conference on Data Engineering, ICDE*, 2023.

[54] H. Hemati, L. Pellegrini, X. Duan, Z. Zhao, F. Xia, M. Masana, B. Tscheschner, E. Veas, Y. Zheng, S. Zhao, S.-Y. Li, S.-J. Huang, V. Lomonaco, and G. M. van de Ven, "Continual learning in the presence of repetition," *Neural Netw.*, vol. 183, no. C, Feb. 2025.

[55] S. Zhang, O. Jafari, and P. Nagarkar, "A survey on machine learning techniques for auto labeling of video, audio, and text data," 2021.

[56] Y. Wu, K. Sharma, C. Seah, and S. Zhang, "Sentistream: A co-training framework for adaptive online sentiment analysis in evolving data streams," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 6198–6212.

[57] H. Guo, H. Li, Q. Ren, and W. Wang, "Concept drift type identification based on multi-sliding windows," *Information Sciences*, vol. 585, pp. 1–23, 2022.

[58] Z. Mai, R. Li, J. Jeong, D. Quispe, H. Kim, and S. Sanner, "Online continual learning in image classification: An empirical survey," pp. 28–51, 2022.

[59] J. Zhang, D. Guo, Y. Wu, X. Xu, and H. Liu, "Toward lifelong learning for industrial defect classification: A proposed framework," *IEEE Robotics & Automation Magazine*, 2023.

[60] C. Tomani, S. Gruber, M. E. Erdem, D. Cremers, and F. Buettner, "Post-hoc uncertainty calibration for domain drift scenarios," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 10 124–10 132.

[61] C.-C. Lin, D.-J. Deng, C.-H. Kuo, and L. Chen, "Concept drift detection and adaption in big imbalance industrial iot data using an ensemble learning method of offline classifiers," *IEEE Access*, vol. 7, pp. 56 198–56 207, 2019.

[62] H. Sun, Y. Lin, Q. Zou, S. Song, J. Fang, and H. Yu, “Convolutional neural networks based remote sensing scene classification under clear and cloudy environments,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 713–720.

[63] A. Steimers and M. Schneider, “Sources of risk of ai systems,” *International Journal of Environmental Research and Public Health*, vol. 19, no. 6, 2022.

[64] K. Rahmani, R. Thapa, P. Tsou, S. C. Chetty, G. Barnes, C. Lam, and C. F. Tso, “Assessing the effects of data drift on the performance of machine learning models used in clinical sepsis prediction,” *International Journal of Medical Informatics*, vol. 173, p. 104930, 2023.

[65] S. Farquhar and Y. Gal, “What ’out-of-distribution’ is and is not,” in *NeurIPS ML Safety Workshop*, 2022.

[66] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, “Data management challenges in production machine learning,” in *Proceedings of the 2017 ACM international conference on management of data*, 2017, pp. 1723–1726.

[67] Z. Li, X. Xu, T. Shen, C. Xu, J.-C. Gu, Y. Lai, C. Tao, and S. Ma, “Leveraging large language models for nlg evaluation: Advances and challenges,” *arXiv preprint arXiv:2401.07103*, 2024.

[68] J.-Y. Yao, K.-P. Ning, Z.-H. Liu, M.-N. Ning, Y.-Y. Liu, and L. Yuan, “Llm lies: Hallucinations are not bugs, but features as adversarial examples,” *arXiv preprint arXiv:2310.01469*, 2023.

[69] C. K. Tantithamthavorn, F. Palomba, F. Khomh, and J. J. Chua, “Mlops, llmops, fmops, and beyond,” *IEEE Software*, vol. 42, no. 01, pp. 26–32, 2025.

[70] A. König, P. Windirsch, M. Gasteier, and M. Glesner, “Visual inspection in industrial manufacturing,” *IEEE Micro*, vol. 15, no. 3, p. 26–31, Jun. 1995.

[71] A. Thomas, M. Rodd, J. Holt, and C. Neill, “Real-time industrial visual inspection: A review,” *Real-Time Imaging*, vol. 1, no. 2, pp. 139–158, 1995.

[72] A. Agarwal, A. Ajith, C. Wen, V. Stryzheus, B. Miller, M. Chen, M. K. Johnson, J. L. S. Rincon, J. Rosca, and W. Yuan, “Robotic defect inspection with visual and tactile perception for large-scale components,” in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2023, pp. 10 110–10 116.

[73] J. Aust and D. Pons, "Comparative analysis of human operators and advanced technologies in the visual inspection of aero engine blades," *Applied Sciences*, vol. 12, no. 4, p. 2250, 2022.

[74] M. Mazzetto, M. Teixeira, É. O. Rodrigues, and D. Casanova, "Deep learning models for visual inspection on automotive assembling line," *arXiv preprint arXiv:2007.01857*, 2020.

[75] R. T. Chin and C. A. Harlow, "Automated visual inspection: A survey," *IEEE transactions on pattern analysis and machine intelligence*, no. 6, pp. 557–573, 1982.

[76] L. Zeng, F. Wan, B. Zhang, and X. Zhu, "Automated visual inspection for precise defect detection and classification in cbn inserts," *Sensors*, vol. 24, no. 23, 2024.

[77] H. Mende, A. Peters, F. Ibrahim, and R. H. Schmitt, "Integrating deep learning and rule-based systems into a smart devices decision support system for visual inspection in production," *Procedia CIRP*, vol. 109, pp. 305–310, 2022, 32nd CIRP Design Conference (CIRP Design 2022) - Design in a changing world.

[78] M. Sharma, J. Lim, and H. Lee, "The amalgamation of the object detection and semantic segmentation for steel surface defect detection," *Applied Sciences*, vol. 12, no. 12, 2022.

[79] S. Deshpande, V. Venugopal, M. Kumar, and S. Anand, "Deep learning-based image segmentation for defect detection in additive manufacturing: An overview," *The International Journal of Advanced Manufacturing Technology*, vol. 134, no. 5, pp. 2081–2105, 2024.

[80] J. Lee, Y. C. Lee, and J. T. Kim, "Fault detection based on one-class deep learning for manufacturing applications limited to an imbalanced database," *Journal of Manufacturing Systems*, vol. 57, pp. 357–366, 2020.

[81] P. Bergmann, K. Batzner, M. Fauser, D. Sattlegger, and C. Steger, "The MVTec anomaly detection dataset: A comprehensive real-world dataset for unsupervised anomaly detection," *International Journal of Computer Vision*, vol. 129, no. 4, pp. 1038–1059, 04 2021.

[82] 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*, vol. 1, no. 2, 2023.

[83] Y. Li, S. Wang, H. Ding, and H. Chen, "Large language models in finance: A survey," in *Proceedings of the fourth ACM international conference on AI in finance*, 2023, pp. 374–382.

[84] L. Chen, Y. Lei, S. Jin, Y. Zhang, and L. Zhang, "Rlingua: Improving reinforcement learning sample efficiency in robotic manipulations with large language models," *IEEE Robotics and Automation Letters*, 2024.

[85] S. Rasnayaka, G. Wang, R. Shariffdeen, and G. N. Iyer, "An empirical study on usage and perceptions of llms in a software engineering project," in *Proceedings of the 1st International Workshop on Large Language Models for Code*, 2024, pp. 111–118.

[86] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, 2025.

[87] S. Dai, C. Xu, S. Xu, L. Pang, Z. Dong, and J. Xu, "Bias and unfairness in information retrieval systems: New challenges in the llm era," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024, pp. 6437–6447.

[88] 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.

[89] 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.

[90] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[91] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love *et al.*, "Gemma: Open models based on gemini research and technology," *arXiv preprint arXiv:2403.08295*, 2024.

[92] C. Team, H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo, A. Hu, C. A. Choquette-Choo, J. Shen, J. Kelley *et al.*, "Codegemma: Open code models based on gemma," *arXiv preprint arXiv:2406.11409*, 2024.

[93] EvalPlus Team. (2025) Evalplus leaderboard. Available online.

[94] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[95] OpenAI, "Gpt-4o mini: Advancing cost-efficient intelligence," 2024, Available online.

[96] OpenAI, "Gpt-3.5 model documentation," 2023, Available online.

[97] C. Ling, X. Zhao, J. Lu, C. Deng, C. Zheng, J. Wang, T. Chowdhury, Y. Li, H. Cui, X. Zhang *et al.*, "Domain specialization as the key to make large language models disruptive: A comprehensive survey," *arXiv preprint arXiv:2305.18703*, 2023.

[98] M. Dehghan, J. J. Wu, F. H. Fard, and A. Ouni, "Mergerepair: An exploratory study on merging task-specific adapters in code llms for automated program repair," *arXiv preprint arXiv:2408.09568*, 2024.

[99] P. Yadav, D. Tam, L. Choshen, C. A. Raffel, and M. Bansal, "Ties-merging: Resolving interference when merging models," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[100] E. Yang, L. Shen, G. Guo, X. Wang, X. Cao, J. Zhang, and D. Tao, "Model merging in llms, mllms, and beyond: Methods, theories, applications and opportunities," *arXiv preprint arXiv:2408.07666*, 2024.

[101] W. Lu, R. K. Luu, and M. J. Buehler, "Fine-tuning large language models for domain adaptation: Exploration of training strategies, scaling, model merging and synergistic capabilities," *arXiv preprint arXiv:2409.03444*, 2024.

[102] 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.

[103] Y. Yu, G. Rong, H. Shen, H. Zhang, D. Shao, M. Wang, Z. Wei, Y. Xu, and J. Wang, "Fine-tuning large language models to improve accuracy and comprehensibility of automated code review," *ACM transactions on software engineering and methodology*, vol. 34, no. 1, pp. 1–26, 2024.

[104] Y.-D. Tsai, M. Liu, and H. Ren, "Code less, align more: Efficient llm fine-tuning for code generation with data pruning," *arXiv preprint arXiv:2407.05040*, 2024.

[105] C. Feutry, P. Piantanida, F. Alberge, and P. Duhamel, “A simple statistical method to detect covariate shift,” in *GRETSI 2019-XXVIIème Colloque francophone de traitement du signal et des images*, 2019.

[106] F. Alberge, C. Feutry, P. Duhamel, and P. Piantanida, “Detecting covariate shift with black box predictors,” in *2019 26th International Conference on Telecommunications (ICT)*. IEEE, 2019, pp. 324–329.

[107] T. Ginsberg, Z. Liang, and R. G. Krishnan, “A learning based hypothesis test for harmful covariate shift,” *arXiv preprint arXiv:2212.02742*, 2022.

[108] K. Lee, H. Lee, K. Lee, and J. Shin, “Training confidence-calibrated classifiers for detecting out-of-distribution samples,” *arXiv preprint arXiv:1711.09325*, 2017.

[109] H. d. M. d. Bourboux, “Trustgan: Training safe and trustworthy deep learning models through generative adversarial networks,” *arXiv preprint arXiv:2211.13991*, 2022.

[110] I. A. Nikolov, M. P. Philipsen, J. Liu, J. V. Dueholm, A. S. Johansen, K. Nasrollahi, and T. B. Moeslund, “Seasons in drift: A long-term thermal imaging dataset for studying concept drift,” in *Thirty-fifth Conference on Neural Information Processing Systems*. Neural Information Processing Systems Foundation, 2021.

[111] D. Ugrenovic, J. Vankeirsbilck, D. Vanoost, C. R. Kancharla, H. Hallez, T. Holvoet, and J. Boydens, “Towards classification trustworthiness: one-class classifier ensemble,” in *2021 XXX International Scientific Conference Electronics (ET)*. IEEE, 2021, pp. 1–6.

[112] M. Testi, M. Ballabio, E. Frontoni, G. Iannello, S. Moccia, P. Soda, and G. Vessio, “Mlops: a taxonomy and a methodology,” *IEEE Access*, vol. 10, pp. 63 606–63 618, 2022.

[113] Y. Lyu, H. Li, Z. M. Jiang, and A. E. Hassan, “On the model update strategies for supervised learning in aiops solutions,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–38, 2024.

[114] A. Paleyes, R.-G. Urma, and N. D. Lawrence, “Challenges in deploying machine learning: a survey of case studies,” *ACM computing surveys*, vol. 55, no. 6, pp. 1–29, 2022.

[115] L. H. Clemmensen and R. D. Kjærsgaard, “Data representativity for machine learning and ai systems,” *arXiv preprint arXiv:2203.04706*, 2022.

[116] J. Encinas, A. Rodríguez, and A. Otero, “Leveraging incremental machine learning for reconfigurable systems modeling under dynamic workloads,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 18, no. 1, pp. 1–27, 2025.

[117] L. Liu, Y. Wang, T. Wang, D. Guan, J. Wu, J. Chen, R. Xiao, W. Zhu, and F. Fang, “Continual transfer learning for cross-domain click-through rate prediction at taobao,” in *Companion Proceedings of the ACM Web Conference 2023*, 2023, pp. 346–350.

[118] Y. Li, X. Yang, H. Wang, X. Wang, and T. Li, “Learning to prompt knowledge transfer for open-world continual learning.” AAAI Press, 2024.

[119] Fabforce GmbH & Co., “Fabforce gmbh & co.” https://www.fabforce.com/, 2025, accessed: 2025-03-26.

[120] M. Banf and G. Steinhagen, “Who supervises the supervisor? model monitoring in production using deep feature embeddings with applications to workpiece inspection,” *arXiv preprint arXiv:2201.06599*, 2022.

[121] W. Sun, R. Al Kontar, J. Jin, and T.-S. Chang, “A continual learning framework for adaptive defect classification and inspection,” *Journal of Quality Technology*, vol. 55, no. 5, pp. 598–614, 2023.

[122] A. Cordier, D. Das, and P. Gutierrez, “Active learning using weakly supervised signals for quality inspection,” in *Fifteenth International Conference on Quality Control by Artificial Vision*, vol. 11794. SPIE, 2021, pp. 278–285.

[123] S. Dou, H. Jia, S. Wu, H. Zheng, W. Zhou, M. Wu, M. Chai, J. Fan, C. Huang, Y. Tao *et al.*, “What’s wrong with your code generated by large language models? an extensive study,” *arXiv preprint arXiv:2407.06153*, 2024.

[124] W. C. Ouédraogo, Y. Li, K. Kaboré, X. Tang, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyandé, “Test smells in llm-generated unit tests,” *arXiv preprint arXiv:2410.10628*, 2024.

[125] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, “Security weaknesses of copilot generated code in github,” *arXiv preprint arXiv:2310.02059*, 2023.

[126] T. Coignion, C. Quinton, and R. Rouvoy, “A performance study of llm-generated code on leetcode,” in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 79–89.

[127] F. Tambon, A. Moradi-Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and G. Antoniol, "Bugs in large language models generated code: An empirical study," *Empirical Software Engineering*, vol. 30, no. 3, pp. 1–48, 2025.

[128] 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.

[129] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," *arXiv preprint arXiv:2308.01861*, 2023.

[130] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pretrained models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[131] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul *et al.*, "Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions," *arXiv preprint arXiv:2406.15877*, 2024.

[132] M. Du, A. T. Luu, B. Ji, and S.-K. Ng, "Mercury: An efficiency benchmark for llm code synthesis," *arXiv preprint arXiv:2402.07844*, 2024.

[133] D. Huang, Y. Qing, W. Shang, H. Cui, and J. M. Zhang, "Effibench: Benchmarking the efficiency of automatically generated code," *arXiv preprint arXiv:2402.02037*, 2024.

[134] 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.

[135] 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.

[136] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[137] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression test selection techniques: A survey," *Informatica*, vol. 35, no. 3, 2011.

[138] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of chatgpt in code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–28, 2025.

[139] T. Dixit, D. Lee, S. Fang, S. S. Harsha, A. Sureshan, A. Maharaj, and Y. Li, "Retain: Interactive tool for regression testing guided llm migration," *arXiv preprint arXiv:2409.03928*, 2024.

[140] Y.-D. Tsai, M. Liu, and H. Ren, "Code less, align more: Efficient llm fine-tuning for code generation with data pruning," 2024.

[141] S. Liu, H. Wu, B. He, X. Han, M. Yuan, and L. Song, "Sens-merging: Sensitivity-guided parameter balancing for merging large language models," 2025.

[142] M. Elhoseny, R. Ayadi, R. M. Abd El-Aziz, A. I. Taloba, H. Aljuaid, N. O. Hamed, and M. A. Khder, "Deep learning-based soft sensors for improving the flexibility for automation of industry," *Wireless Communications and Mobile Computing*, vol. 2022, p. 5450473, 2022.

[143] O. Semeniuta, S. Dransfeld, K. Martinsen, and P. Falkman, "Towards increased intelligence and automatic improvement in industrial vision systems," *Procedia cirp*, vol. 67, pp. 256–261, 2018.

[144] J. Villalba-Diez, D. Schmidt, R. Gevers, J. Ordieres-Meré, M. Buchwitz, and W. Wellbrock, "Deep learning for industrial computer vision quality control in the printing industry 4.0," *Sensors*, vol. 19, no. 18, 2019.

[145] X. Zheng, S. Zheng, Y. Kong, and J. Chen, "Recent advances in surface defect inspection of industrial products using deep learning techniques," *The International Journal of Advanced Manufacturing Technology*, vol. 113, no. 1, pp. 35–58, 2021.

[146] M. A. Hashmani, S. M. Jameel, H. Alhussain, M. Rehman, and A. Budiman, "Accuracy performance degradation in image classification models due to concept drift," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 5, 2019.

[147] A. Kahraman, M. Kantardzic, and M. Kotan, "Dynamic modeling with integrated concept drift detection for predicting real-time energy consumption of industrial machines," *IEEE Access*, vol. 10, pp. 104 622–104 635, 2022.

[148] H. Hu, M. Kantardzic, and T. S. Sethi, "No free lunch theorem for concept drift detection in streaming data classification: a review," *WIREs Data Mining and Knowledge Discovery*, vol. 10, 2019.

[149] A. Mallick, K. Hsieh, B. Arzani, and G. Joshi, "Matchmaker: Data drift mitigation in machine learning for large-scale systems," in *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4, 2022, pp. 77–94.

[150] B. Chen, J. Jiang, X. Wang, P. Wan, J. Wang, and M. Long, "Debiased self-training for semi-supervised learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 32 424–32 437, 2022.

[151] F. Bayram, B. S. Ahmed, and A. Kassler, "From concept drift to model degradation: An overview on performance-aware drift detectors," *Knowledge-Based Systems*, vol. 245, p. 108632, 2022.

[152] G. Kim, C. Xiao, T. Konishi, Z. Ke, and B. Liu, "A theoretical study on solving continual learning," pp. 5065–5079, 2022.

[153] B. Sahiner, W. Chen, R. K. Samala, and N. Petrick, "Data drift in medical machine learning: implications and potential remedies," *British Journal of Radiology*, vol. 96, no. 1150, p. 20220878, 03 2023.

[154] N. Polyzotis, M. Zinkevich, S. Roy, E. Breck, and S. Whang, "Data validation for machine learning," *Proceedings of machine learning and systems*, vol. 1, pp. 334–347, 2019.

[155] S. Vernekar, A. Gaurav, V. Abdelzad, T. Denouden, R. Salay, and K. Czarnecki, "Out-of-distribution detection in classifiers via generation," *arXiv preprint arXiv:1910.04241*, 2019.

[156] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning." MIT Press, 2016.

[157] T. Li, A. Beirami, M. Sanjabi, and V. Smith, "Tilted empirical risk minimization," in *International Conference on Learning Representations*, 2021.

[158] L. Yuan, H. S. Park, and E. Lejeune, "Towards out of distribution generalization for problems in mechanics," *Computer Methods in Applied Mechanics and Engineering*, vol. 400, p. 115569, 2022.

[159] M. Zhang, J. Yuan, Y. He, W. Li, Z. Chen, and K. Kuang, “Map: Towards balanced generalization of iid and ood through model-agnostic adapters,” in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 11 921–11 931.

[160] K. Kammerer, B. Hoppenstedt, R. Pryss, S. Stökler, J. Allgaier, and M. Reichert, “Anomaly detections for manufacturing systems based on sensor data—insights into two challenging real-world production settings,” *Sensors*, vol. 19, no. 24, p. 5370, 2019.

[161] T. Le, T. Miller, R. Singh, and L. Sonenberg, “Explaining model confidence using counterfactuals,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 10, 2023, pp. 11 856–11 864.

[162] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, “On calibration of modern neural networks,” in *International conference on machine learning.* PMLR, 2017, pp. 1321–1330.

[163] S. Desai and G. Durrett, “Calibration of pre-trained transformers,” *arXiv preprint arXiv:2003.07892*, 2020.

[164] D. Guillory, V. Shankar, S. Ebrahimi, T. Darrell, and L. Schmidt, “Predicting with confidence on unseen distributions,” in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 1134–1144.

[165] T. Silva Filho, H. Song, M. Perello-Nieto, R. Santos-Rodriguez, M. Kull, and P. Flach, “Classifier calibration: a survey on how to assess and improve predicted class probabilities,” *Machine Learning*, vol. 112, no. 9, pp. 3211–3260, 2023.

[166] T. Ye, Z. Li, J. Wang, N. Cheng, and J. Xiao, “Efficient uncertainty estimation with gaussian process for reliable dialog response retrieval,” in *ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE, 2023, pp. 1–5.

[167] H. Wei, R. Xie, H. Cheng, L. Feng, B. An, and Y. Li, “Mitigating neural network overconfidence with logit normalization,” in *International conference on machine learning.* PMLR, 2022, pp. 23 631–23 644.

[168] Y. Ovadia, E. Fertig, J. Ren, Z. Nado, D. Sculley, S. Nowozin, J. Dillon, B. Lakshminarayanan, and J. Snoek, “Can you trust your model’s uncertainty? evaluating predictive uncertainty under dataset shift,” *Advances in neural information processing systems*, vol. 32, 2019.

[169] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space," in *Database Theory—ICDT 2001: 8th International Conference London, UK, January 4–6, 2001 Proceedings 8*. Springer, 2001, pp. 420–434.

[170] W. Yu, M. Zhang, and Y. Shen, "Spatial revising variational autoencoder-based feature extraction method for hyperspectral images," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 59, no. 2, pp. 1410–1423, 2020.

[171] S. Khalid, T. Khalil, and S. Nasreen, "A survey of feature selection and feature extraction techniques in machine learning," in *2014 science and information conference*. IEEE, 2014, pp. 372–378.

[172] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[173] F. Ye and A. G. Bors, "Learning joint latent representations based on information maximization," *Information Sciences*, vol. 567, pp. 216–236, 2021.

[174] J. Lucas, G. Tucker, R. B. Grosse, and M. Norouzi, "Don't blame the elbo! a linear vae perspective on posterior collapse," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[175] M. S. Seyfioglu, K. Bouyarmane, S. Kumar, A. Tavanaei, and I. B. Tutar, "Diffuse to choose: Enriching image conditioned inpainting in latent diffusion models for virtual try-all," *arXiv preprint arXiv:2401.13795*, 2024.

[176] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *Ieee Access*, vol. 6, pp. 14 410–14 430, 2018.

[177] H. Hu, M. Kantardzic, and T. S. Sethi, "No free lunch theorem for concept drift detection in streaming data classification: A review," *WIREs Data Mining and Knowledge Discovery*, vol. 10, no. 2, p. e1327, 2020.

[178] P. Lavergne and V. Patilea, "Breaking the curse of dimensionality in nonparametric testing," *Journal of Econometrics*, vol. 143, no. 1, pp. 103–122, 2008.

[179] A. Farzad and T. A. Gulliver, "Unsupervised log message anomaly detection," *ICT Express*, vol. 6, no. 3, pp. 229–237, 2020.

[180] S. S. Khan and M. G. Madden, "One-class classification: taxonomy of study and review of techniques," *The Knowledge Engineering Review*, vol. 29, no. 3, pp. 345–374, 2014.

[181] D. Ugrenovic, J. Vankeirsbilck, D. Vanoost, C. R. Kancharla, H. Hallez, T. Holvoet, and J. Boydens, “Towards classification trustworthiness: one-class classifier ensemble,” in *2021 XXX International Scientific Conference Electronics (ET)*, 2021, pp. 1–6.

[182] C. Shorten and T. M. Khoshgoftaar, “A survey on image data augmentation for deep learning,” *Journal of big data*, vol. 6, no. 1, pp. 1–48, 2019.

[183] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, “Deephunter: a coverage-guided fuzz testing framework for deep neural networks,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 146–157.

[184] H. B. Braiek and F. Khomh, “Deepevolution: A search-based testing approach for deep neural networks,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 454–458.

[185] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[186] T. Pearce, A. Brintrup, and J. Zhu, “Understanding softmax confidence and uncertainty,” *arXiv preprint arXiv:2106.04972*, 2021.

[187] R. Gontijo-Lopes, S. J. Smullin, E. D. Cubuk, and E. Dyer, “Affinity and diversity: Quantifying mechanisms of data augmentation,” *arXiv preprint arXiv:2002.08973*, 2020.

[188] M. R. Lyu, B. Ray, A. Roychoudhury, S. H. Tan, and P. Thongtanunam, “Automatic programming: Large language models and beyond,” *ACM Transactions on Software Engineering and Methodology*, 2024.

[189] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[190] Z. Zheng, K. Ning, Q. Zhong, J. Chen, W. Chen, L. Guo, W. Wang, and Y. Wang, “Towards an understanding of large language models in software engineering tasks,” *Empirical Software Engineering*, vol. 30, no. 2, p. 50, 2025.

[191] C. Michelutti, J. Eckert, M. Monecke, J. Klein, and S. Glesner, "A systematic study on the potentials and limitations of llm-assisted software development," in *2024 2nd International Conference on Foundation and Large Language Models (FLLM)*. IEEE, 2024, pp. 330–338.

[192] J. Wang and Y. Chen, "A review on code generation with llms: Application and evaluation," in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, 2023, pp. 284–289.

[193] L. Zhong and Z. Wang, "Can llm replace stack overflow? a study on robustness and reliability of large language model code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 19, 2024, pp. 21 841–21 849.

[194] M. Dunne, K. Schram, and S. Fischmeister, "Weaknesses in llm-generated code for embedded systems networking," in *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2024, pp. 250–261.

[195] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on software engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[196] A. A. Abbassi, L. Da Silva, A. Nikanjam, and F. Khomh, "Replication package for: Unveiling inefficiencies in llm-generated code: Toward a comprehensive taxonomy," Available on GitHub, 2025.

[197] IEEE Spectrum, "Top programming languages 2024," Available online, 2024.

[198] 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.

[199] F. Lin, D. J. Kim *et al.*, "When llm-based code generation meets the software development process," *arXiv preprint arXiv:2403.15852*, 2024.

[200] T. Miah and H. Zhu, "User centric evaluation of code generation tools," in *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE, 2024, pp. 109–119.

[201] Meta AI, "Codellama: Open foundation models for code," Available on Hugging Face, 2024.

[202] Google DeepMind, "Codegemma: Open code models based on gemma," Available on Hugging Face, 2024.

[203] DeepSeek AI, “Deepseek-coder: Open-source code language models,” Available on Hugging Face, 2024.

[204] T. Zheng, G. Zhang, T. Shen, X. Liu, B. Y. Lin, J. Fu, W. Chen, and X. Yue, “Opencodeinterpreter: Integrating code generation with execution and refinement,” *arXiv preprint arXiv:2402.14658*, 2024.

[205] Y. Zhao, Z. Luo, Y. Tian, H. Lin, W. Yan, A. Li, and J. Ma, “Codejudge-eval: Can large language models be good judges in code understanding?” *arXiv preprint arXiv:2408.10718*, 2024.

[206] Z. Sollenberger, J. Patel, C. Munley, A. Jarmusch, and S. Chandrasekaran, “Llm4vv: Exploring llm-as-a-judge for validation and verification testsuites,” in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 2024, pp. 1885–1893.

[207] T. Wang, M. Tao, R. Fang, H. Wang, S. Wang, Y. E. Jiang, and W. Zhou, “Ai persona: Towards life-long personalization of llms,” *arXiv preprint arXiv:2412.13103*, 2024.

[208] A. H. Yahmed, A. A. Abbassi, A. Nikanjam, H. Li, and F. Khomh, “Deploying deep reinforcement learning systems: a taxonomy of challenges,” in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2023, pp. 26–38.

[209] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, “Taxonomy of real faults in deep learning systems,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.* New York, NY, USA: Association for Computing Machinery, 2020, p. 1110–1121.

[210] M. X. Liu, A. Sarkar, C. Negreanu, B. Zorn, J. Williams, N. Toronto, and A. D. Gordon, ““what it wants me to say”: Bridging the abstraction gap between end-user programmers and code-generating large language models,” in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems.* New York, NY, USA: Association for Computing Machinery, 2023.

[211] J. Cohen, “A coefficient of agreement for nominal scales,” *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[212] D. Spadini, M. Aniche, and A. Bacchelli, “Pydriller: Python framework for mining software repositories,” in *Proceedings of the 2018 26th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 908–911.

[213] 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.

[214] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, vol. 167, p. 110610, 2020.

[215] W. Takerngsaksiri, M. Fu, C. Tantithamthavorn, J. Pasuksmit, K. Chen, and M. Wu, "Code readability in the age of large language models: An industrial case study from atlassian," *arXiv preprint arXiv:2501.11264*, 2025.

[216] H. Nunes, E. Figueiredo, L. Rocha, S. Nadi, F. Ferreira, and G. Esteves, "Evaluating the effectiveness of llms in fixing maintainability issues in real-world projects," *arXiv preprint arXiv:2502.02368*, 2025.

[217] F. Xu, Q. Lin, J. Han, T. Zhao, J. Liu, and E. Cambria, "Are large language models really good logical reasoners? a comprehensive evaluation and beyond," *IEEE Transactions on Knowledge and Data Engineering*, 2025.

[218] S. Fischer, G. K. Michelon, W. K. Assunção, R. Ramler, and A. Egyed, "Designing a test model for a configurable system: An exploratory study of preprocessor directives and feature toggles," in *Proceedings of the 17th International Working Conference on Variability Modelling of Software-Intensive Systems*, 2023, pp. 31–39.

[219] C. Wang, K. Huang, J. Zhang, Y. Feng, L. Zhang, Y. Liu, and X. Peng, "How and why llms use deprecated apis in code completion? an empirical study," *arXiv preprint arXiv:2406.09834*, 2024.

[220] V. Nardone, B. Muse, M. Abidi, F. Khomh, and M. Di Penta, "Video game bad smells: What they are and how developers perceive them," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–35, 2023.

[221] A. Nikanjam, M. M. Morovati, F. Khomh, and H. Ben Braiek, "Faults in deep reinforcement learning programs: a taxonomy and a detection approach," *Automated software engineering*, vol. 29, no. 1, p. 8, 2022.

[222] A. Serban and J. Visser, "Adapting software architectures to machine learning challenges," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 152–163.

[223] R. Tian, Y. Ye, Y. Qin, X. Cong, Y. Lin, Y. Pan, Y. Wu, H. Hui, W. Liu, Z. Liu *et al.*, "Debugbench: Evaluating debugging capability of large language models," *arXiv preprint arXiv:2401.04621*, 2024.

[224] D. Oliveira, "Recommending code understandability improvements based on code reviews," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 2021, pp. 131–132.

[225] Microsoft Corporation, "Microsoft copilot," Available online, 2024.

[226] Anysphere Inc., "Cursor ai: Ai-powered code editor," Available online, 2024.

[227] B. Szalontai, A. Vadász, T. Márton, B. Pintér, and T. Gregorics, "Fine-tuning codellama to fix bugs," in *Proceedings of International Conference on Recent Innovations in Computing*, Z. Illés, C. Verma, P. J. S. Gonçalves, and P. K. Singh, Eds. Singapore: Springer Nature Singapore, 2024, pp. 497–509.

[228] B. Lei, Y. Li, and Q. Chen, "Autocoder: Enhancing code large language model with\textsc {AIEV-Instruct}," *arXiv preprint arXiv:2405.14906*, 2024.

[229] B. Poudel, A. Cook, S. Traore, and S. Ameli, "Documint: Docstring generation for python using small language models," *arXiv preprint arXiv:2405.10243*, 2024.

[230] J. Li, A. Sangalay, C. Cheng, Y. Tian, and J. Yang, "Fine tuning large language model for secure code generation," in *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, 2024, pp. 86–90.

[231] M. L. Siddiq, L. Roney, J. Zhang, and J. C. D. S. Santos, "Quality assessment of chatgpt generated code and their use by developers," in *Proceedings of the 21st International Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 2024, p. 152–156.

[232] A. A. Abbassi, L. Da Silva, A. Nikanjam, and F. Khomh, "Unveiling inefficiencies in llm-generated code: Toward a comprehensive taxonomy," *arXiv preprint arXiv:2503.06327*, 2025.

[233] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[234] H. B. Hassan, Q. I. Sarhan, and Á. Beszédes, "Evaluating python static code analysis tools using fair principles," *IEEE Access*, vol. 12, pp. 173 647–173 659, 2024.

[235] Pylint, "Pylint: Python code static checker," Available online, 2025, accessed: 2025-01-23.

[236] L. Traini, D. Di Pompeo, M. Tucci, B. Lin, S. Scalabrino, G. Bavota, M. Lanza, R. Oliveto, and V. Cortellessa, "How software refactoring impacts execution time," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–23, 2021.

[237] Hugging Face, "Hugging face hub," Available online, 2025.

[238] A. A. Abbassi, L. Da Silva, A. Nikanjam, and F. Khomh, "Recatcher: Towards regression testing for llms in code generation – replication package," Available online, 2025.

[239] P. W. O'Hearn, "Incorrectness logic," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, 2019.

[240] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 177–187.

[241] R. M. a. dos Santos and M. A. Gerosa, "Impacts of coding practices on readability," in *Proceedings of the 26th Conference on Program Comprehension.* New York, NY, USA: Association for Computing Machinery, 2018, p. 277–285.

[242] D. Cotroneo, A. Foggia, C. Improta, P. Liguori, and R. Natella, "Automating the correctness assessment of ai-generated code for security contexts," *Journal of Systems and Software*, p. 112113, 2024.

[243] J. P. Faria and R. Abreu, "Case studies of development of verified programs with dafny for accessibility assessment," in *International Conference on Fundamentals of Software Engineering.* Springer, 2023, pp. 25–39.

[244] J. Törnblom and S. Nadjm-Tehrani, "Scaling up memory-efficient formal verification tools for tree ensembles," *arXiv preprint arXiv:2105.02595*, 2021.

[245] M. Olan, "Unit testing: test early, test often," *J. Comput. Sci. Coll.*, vol. 19, no. 2, p. 319–328, Dec. 2003.

[246] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, "Unit test generation using generative ai: A comparative performance analysis of autogeneration tools," in *Proceedings of the 1st International Workshop on Large Language Models for Code*, 2024, pp. 54–61.

[247] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 951–971, 2024.

[248] J. Novak, A. Krajnc *et al.*, "Taxonomy of static code analysis tools," in *The 33rd international convention MIPRO.* IEEE, 2010, pp. 418–422.

[249] I. Gomes, P. Morgado, T. Gomes, and R. Moreira, "An overview on the static code analysis approach in software development," *Faculdade de Engenharia da Universidade do Porto, Portugal*, vol. 16, 2009.

[250] D. Singh, V. R. Sekar, K. T. Stolee, and B. Johnson, "Evaluating how static analysis tools can reduce code review effort," in *2017 IEEE symposium on visual languages and human-centric computing (VL/HCC).* IEEE, 2017, pp. 101–105.

[251] SonarSource, "Sonarqube: Continuous code quality and security," Available online, 2025.

[252] PMD Developers, "Pmd: An extensible cross-language static code analyzer," https://docs.pmd-code.org/latest/index.html, accessed: 2025-01-23.

[253] S. Habchi, X. Blanc, and R. Rouvoy, "On adopting linters to deal with performance concerns in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 6–16.

[254] W. Rafnsson, R. Giustolisi, M. Kragerup, and M. Høyrup, "Fixing vulnerabilities automatically with linters," in *Network and System Security: 14th International Conference, NSS 2020, Melbourne, VIC, Australia, November 25–27, 2020, Proceedings 14.* Springer, 2020, pp. 224–244.

[255] H. Gulabovska and Z. Porkoláb, "Survey on static analysis tools of python programs." in *SQAMIA*, 2019.

[256] A. Groce, I. Ahmed, J. Feist, G. Grieco, J. Gesi, M. Meidani, and Q. Chen, "Evaluating and improving static analysis tools via differential mutation analysis," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS).* IEEE, 2021, pp. 207–218.

[257] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh, "A systematic literature review on source code similarity measurement and clone detection:

Techniques, applications, and challenges," *Journal of Systems and Software*, p. 111796, 2023.

[258] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2015, pp. 131–140.

[259] H. Cui, M. Xie, T. Su, C. Zhang, and S. H. Tan, "An empirical study of false negatives and positives of static code analyzers from the perspective of historical issues," *arXiv preprint arXiv:2408.13855*, 2024.

[260] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[261] A. Bergel, F. Banados, R. Robbes, and D. Röthlisberger, "Spy: A flexible code profiling framework," *Computer Languages, Systems & Structures*, vol. 38, no. 1, pp. 16–28, 2012.

[262] A.-T. P. Nguyen, V.-D. Hoang *et al.*, "Development of code evaluation system based on abstract syntax tree," *Journal of Technical Education Science*, vol. 19, no. Special Issue 01, pp. 15–24, 2024.

[263] Python Software Foundation, "tracemalloc — trace memory allocations," Available online, 2025.

[264] JetBrains, "Kexercises," Available on Hugging Face, 2021.

APPENDIX A CO-AUTHORSHIP

The research studies in this thesis were submitted/published as follows:

- Trimming the Risk: Towards Reliable Continuous Training for Deep Learning Inspection Systems, **Altaf Allah Abbassi**, Houssem Ben Braiek, Foutse Khomh, Thomas Reid to Elsevier *Journal of Software and Systems*
- Taxonomy of Inefficiencies in LLM-Generated Code, **Altaf Allah Abbassi**, Leuson Da Silva, Amin Nikanjam, Foutse Khomh to *41st IEEE International Conference on Software Maintenance and Evolution (ICSME 2025)*
- ReCatcher: Towards LLMs Regression Testing for Code Generation, **Altaf Allah Abbassi**, Leuson Da Silva, Amin Nikanjam, Foutse Khomh to *48th EEE/ACM International Conference on Software Engineering (ICSE 2026)*

The following publication was produced during my Master's degree studies but is not directly related to the content of the thesis:

- Deploying deep reinforcement learning systems: A taxonomy of challenges, Ahmed Haj Yahmed, **Altaf Allah Abbassi**, Amin Nikanjam, Heng Li, Foutse Khomh, proceedings of the *39th IEEE International Conference on Software Maintenance and Evolution (ICSME 2023)*