Titre: Title:	Diagnostic automatisé des performances dans les systèmes distribués à l'aide de traces, de métriques système et de modèles de langage de grande taille
Auteur: Author:	Maryam Ekhlasi
Date:	2025
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Ekhlasi, M. (2025). Diagnostic automatisé des performances dans les systèmes distribués à l'aide de traces, de métriques système et de modèles de langage de grande taille [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/67029/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/67029/
Directeurs de recherche: Advisors:	Michel Dagenais, & Maxime Lamothe
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Diagnostic automatisé des performances dans les systèmes distribués à l'aide de traces, de métriques système et de modèles de langage de grande taille

MARYAM EKHLASI
Département de génie informatique et génie logiciel

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

Juillet 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Diagnostic automatisé des performances dans les systèmes distribués à l'aide de traces, de métriques système et de modèles de langage de grande taille

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

Daniel ALOISE, président
Michel DAGENAIS, membre et directeur de recherche
Maxime LAMOTHE, membre et codirecteur de recherche
Heng LI, membre
Weiyi SHANG, membre externe

DÉDICACE

To my parents and my brother,
for their constant support
and countless sacrifices...

REMERCIEMENTS

Je tiens à exprimer ma plus profonde gratitude au professeur Michel Dagenais pour sa compréhension remarquable et pour m'avoir offert l'opportunité de travailler sous sa direction. Ses conseils ont été inestimables, façonnant non seulement ma progression académique, mais aussi ma vision de la vie, en m'enseignant l'importance de l'équilibre entre vie professionnelle et vie personnelle. Grâce à son soutien, j'ai pu trouver une direction claire dans mon parcours doctoral, et les leçons apprises resteront gravées en moi pour toujours. Un sincère et chaleureux merci à lui.

Je remercie également le Professeur Maxime Lamothe pour ses conseils et son soutien, qui m'ont grandement aidée tout au long de mon doctorat.

Un remerciement tout particulier à mes parents et à mon frère pour leur soutien constant, leurs sacrifices, et pour avoir toujours été présents à chaque étape de ma vie. Les mots ne suffisent pas à exprimer toute ma reconnaissance.

À mes collègues de recherche et aux membres de mon laboratoire : merci pour votre aide, vos conseils, vos idées partagées, ainsi que pour toutes les conversations et les rires du quotidien. Ce fut un réel plaisir de travailler avec vous.

À mes amis, merci pour votre soutien et pour m'avoir aidé à garder un équilibre de vie au fil des années — pour tous les rires, les larmes, et tout ce qu'il y a entre les deux.

Je souhaite remercier les professeurs Heng Li, Daniel Aloise et Weiyi Shang d'avoir accepté de faire partie de mon jury, ainsi que pour leurs commentaires précieux et leurs remarques éclairées.

Enfin, je tiens à remercier chaleureusement le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG), Prompt, Ericsson, Ciena, AMD et EfficiOS pour le financement de ce projet.

RÉSUMÉ

Les systèmes distribués évoluent rapidement et suscitent un grand intérêt auprès des grandes entreprises, car ils ne sont pas liés à un langage de programmation spécifique, et sont facilement évolutifs et maintenables. Cependant, en raison de leur architecture complexe et de la communication entre services, identifier un service défaillant en cas de problème ou d'erreur est une tâche extrêmement difficile et chronophage. Les ingénieurs doivent avoir une compréhension approfondie de l'architecture pour localiser précisément l'origine du problème. Pour y faire face, les systèmes de traçage de haut niveau et de bas niveau sont devenus des outils précieux permettant d'identifier les causes de la dégradation des performances. Le traçage suit une requête depuis une couche de haut niveau à travers plusieurs services, tandis que les systèmes de métriques détectent les problèmes de performance au niveau des machines. Utiliser l'une ou l'autre de ces approches, ou une combinaison des deux, présente des avantages et des défis spécifiques. Dans cette recherche, nous avons abordé ces problématiques en trois phases.

Dans la première phase de notre recherche, nous avons développé DTraComp (Distributed Trace Compare), un logiciel à code source libre conçu pour aider les utilisateurs à comparer plusieurs traces d'exécution dans des systèmes distribués ou monolithiques. Cet outil répond à une limitation clé de nombreux outils d'analyse de performance existants : l'absence de fonctionnalité de comparaison. DTraComp facilite l'identification des services responsables de la dégradation des performances en comparant deux groupes de traces de requêtes et en mettant en évidence leurs différences. Le logiciel est intégré à Eclipse Trace Compass, une plateforme largement utilisée pour la visualisation de traces. Il prend en charge divers formats de traces et génère un graphe de flamme différentiel en utilisant une échelle de couleurs allant du blanc au rouge et au bleu, pour montrer l'intensité et la direction des différences d'exécution. Les utilisateurs peuvent également sélectionner un service ou une portion spécifique pour consulter les métriques système de bas niveau, telles que l'utilisation du CPU ou la mémoire, capturées au niveau du noyau. Ce contexte supplémentaire permet d'identifier la cause profonde des dégradations de performance dans les composants distribués. DTraComp a été évalué sur plusieurs scénarios réels et est actuellement utilisé chez Ericsson pour l'analyse des performances.

Dans la deuxième phase de notre projet, nous avons poursuivi notre travail en développant une approche de filtrage intelligent visant à améliorer la précision et l'efficacité des comparaisons de requêtes. Les systèmes distribués traitent de nombreux types de requêtes, chacun

avec des chemins d'exécution et des structures différents, ce qui rend la comparaison directe difficile. Pour résoudre ce problème, nous avons proposé une solution qui identifie d'abord les requêtes suspectes, en se basant sur des modèles communs et des anomalies de performance. Ensuite, notre méthode filtre et regroupe les requêtes similaires en fonction de leurs similitudes structurelles et comportementales. Cela permet des comparaisons plus pertinentes et aide à isoler la cause profonde de la dégradation des performances. En se concentrant uniquement sur les traces pertinentes, et en extrayant des métriques système détaillées pour les services affectés, notre approche réduit l'utilisation de stockage et la charge d'analyse. Cette stratégie sélective rend le processus de comparaison plus évolutif et adapté aux systèmes en conditions réelles.

Dans la phase finale de notre recherche, nous nous sommes concentrés sur l'un des grands défis industriels : l'interprétation efficace des journaux (logs) et des traces. Les entreprises sont souvent confrontées à la tâche longue et fastidieuse d'analyser manuellement de grandes quantités de données de log et de trace pour détecter et comprendre les problèmes système. Pour répondre à cela, nous avons développé une solution automatisée exploitant les modèles de langage de grande taille (LLMs), combinés à l'apprentissage en contexte (in-context learning). Notre méthode évite le besoin fréquent de réentraîner les modèles en s'adaptant aux changements de structure des logs, sans modifier le modèle sous-jacent. Nous avons introduit un format d'entrée inspiré des graphes de flamme, permettant de réduire significativement le nombre de jetons (tokens), diminuant ainsi les coûts et la latence de traitement, tout en maintenant une haute précision. Cette approche permet aux entreprises d'automatiser plus efficacement l'analyse des logs, de réduire les interruptions de service, et de préserver la confidentialité des données, sans avoir à gérer directement une infrastructure informatique à grande échelle.

ABSTRACT

Distributed systems are growing quickly and are widely used by large companies because they are easy to scale, maintain, and do not depend on a specific programming language. However, their complex structure and the way services communicate with each other make it very difficult and time-consuming to find the cause of problems when something goes wrong. Engineers often need a deep understanding of the system to identify which service is causing the issue. One common way to address this challenge is by using high-level tracing and low-level system metrics. High-level tracing tracks how a request moves across different services, while low-level metrics focus on system performance like CPU or memory usage. Each method has its own advantages and limitations, and sometimes they are used together. In this research, we addressed these challenges through three main phases.

In the first phase of our research, we developed DTraComp (Distributed Trace Compare), an open-source framework designed to help users compare multiple execution traces in both distributed and monolithic systems. This tool addresses a key limitation in many existing performance analysis tools—the lack of a comparison feature. DTraComp makes it easier to identify services responsible for performance degradation by comparing two groups of request traces and highlighting their differences. The framework is integrated with Eclipse Trace Compass, a widely used trace visualization platform. It supports various trace formats and generates a differential flame graph, using a color scale from white to red and blue, to show the magnitude and direction of the differences in execution. Users can also select a specific service or span to view low-level system metrics, such as CPU usage and memory, captured at the kernel level. This additional context helps identify the root cause of performance degradation across distributed components. DTraComp has been evaluated on multiple real-world scenarios and is currently in use at Ericsson for performance analysis.

In the second phase of our project, we extended our work by developing a smarter filtering approach to improve the accuracy and efficiency of request comparisons. Distributed systems handle many types of requests, each with different execution paths and structures, which makes direct comparison difficult. To address this, we proposed a solution that first identifies suspicious requests based on common patterns and performance anomalies. Our method then filters and groups similar requests using their structural and behavioral similarities. This enables more meaningful comparisons and helps isolate the root cause of performance degradation. By focusing only on relevant traces and extracting detailed system metrics for the affected services, our approach reduces storage usage and analysis overhead. This

selective strategy makes the comparison process more scalable and practical for use in real-world systems.

In the final phase of our research, we focused on one of the major industrial challenges—efficient interpretation of logs and traces. Companies often struggle with the time-consuming task of manually analyzing large volumes of log and trace data to detect and understand system issues. To address this, we developed an automated solution that leverages large language models (LLMs) combined with in-context learning. Our method reduces the need for frequent model retraining by adapting to changes in log patterns without modifying the underlying model. We introduced a novel flame-graph-inspired input format that significantly reduces the number of tokens, cutting down both token costs and processing latency, while preserving accuracy. This approach allows industries to automate log analysis more effectively, reduce system downtime, and maintain data privacy—without the need to manage large-scale computational infrastructure directly.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Programming Interface (interface de programmation applicative)
AI	Artificial Intelligence
CTF	Common Trace Format
CPU	Centralized Processing Unit
GPU	Graphics Processing Unit
IO	Input Output
JSON	JavaScript Object Notation
LTTng	Linux Trace Toolkit : next generation
LLM	Large Language Models
PID	Process Identifier
RAM	Random Access Memory
RPC	Remote Procedure Call

CHAPITRE 1 INTRODUCTION

Ces dernières années, de nombreux systèmes logiciels ont évolué vers des architectures distribuées. Plutôt que de concevoir des systèmes monolithiques, les développeurs divisent désormais les applications en services plus petits et indépendants afin d'améliorer la scalabilité, la résilience et la maintenabilité. Ces services peuvent s'exécuter sur des machines distinctes, dans des conteneurs, ou même à travers des centres de données distribués géographiquement. Bien que cette modularisation permette un déploiement plus flexible et une utilisation plus efficace des ressources, elle augmente également la complexité de la surveillance et du diagnostic des performances. Dans ces environnements, un seul service lent ou un composant mal configuré peut provoquer des délais en cascade difficiles à retracer jusqu'à leur origine. Cette complexité met en évidence le besoin d'outils avancés pour soutenir l'analyse des causes profondes (root cause analysis) dans les systèmes distribués à grande échelle.

Ce projet de recherche aborde les défis liés à la surveillance des performances, à l'analyse de traces et au diagnostic des pannes dans les applications distribuées. Il examine si les outils actuels sont capables de gérer l'échelle, la diversité et le volume des données générées par les systèmes modernes basés sur des microservices. Le rôle de ce travail est d'identifier les lacunes des outils existants et de proposer de nouvelles solutions qui améliorent la visibilité du comportement du système, réduisent le temps d'analyse et facilitent la détection automatisée des causes profondes. Une première étape consiste à évaluer l'état de l'art des technologies de traçage et des méthodes de visualisation. Cela inclut l'examen de la capacité des outils actuels à intégrer les données de traçage de haut niveau avec les métriques système de bas niveau, ainsi que leur gestion des défis liés à la synchronisation temporelle, au volume des traces et à l'interprétabilité pour les utilisateurs.

Le traçage est une méthode largement acceptée pour observer les performances dans les systèmes distribués. Cependant, les outils traditionnels rencontrent souvent des difficultés en matière de passage à l'échelle, d'interprétation des données et de liaison des informations de trace à la cause racine des problèmes de performance. De nombreux outils existants se concentrent soit sur les intervalles (spans) de haut niveau, soit sur les données système de bas niveau, mais manquent de mécanismes efficaces pour combiner les deux. Cette recherche évalue donc les performances et les limitations des outils modernes de traçage et de logging, en particulier dans des environnements composés d'applications en conteneurs et de microservices à grande échelle. L'objectif est d'identifier dans quelle mesure ces outils soutiennent les développeurs dans la détection et la compréhension des sources de latence à travers les

couches applicatives et système.

Une fois collectées, les données de trace et de logs doivent être présentées de manière significative. La visualisation joue un rôle clé dans ce processus. Les outils actuels offrent des vues de base, mais manquent souvent de fonctionnalités de comparaison, d'exploration contextuelle, ou d'intégration avec des analyses assistées par l'IA. Cette recherche analyse comment ces limitations affectent la capacité des développeurs à interpréter les données de performance et à réagir rapidement aux problèmes. Elle propose également de nouvelles approches, telles que les graphes de flammes différentiels et l'analyse hybride de traces, permettant d'extraire des informations plus exploitables.

La seconde partie de cette thèse décrit les contributions proposées, organisées autour de trois axes applicatifs. Premièrement, de nouvelles techniques sont développées pour permettre la comparaison entre groupes de requêtes distribuées, à l'aide de visualisations de traces enrichies et de métriques au niveau noyau. Deuxièmement, un cadre de traçage hybride adaptatif est proposé, qui collecte sélectivement des métriques système le long du chemin critique afin de réduire la surcharge tout en améliorant la détection des causes profondes. Troisièmement, un cadre d'analyse de logs basé sur des modèles de langage de grande taille (LLMs) est introduit, utilisant des techniques efficaces et respectueuses de la confidentialité pour automatiser l'interprétation de logs en environnement réel. Chacune de ces contributions est validée à travers des études de cas et des collaborations industrielles.

Le cœur de ce travail est d'améliorer la précision, la scalabilité et l'utilisabilité des outils d'analyse de performance pour les systèmes distribués à grande échelle. Après avoir identifié les limites des outils actuels par revue de la littérature et expérimentation, de nouvelles méthodes sont introduites et évaluées. Celles-ci incluent des améliorations algorithmiques ainsi que des fonctionnalités orientées utilisateur, dont certaines ont été implémentées sous forme de correctifs pour des outils open source, ou directement utilisées en contexte industriel.

Plus généralement, cette thèse cherche à répondre aux questions de recherche suivantes : (1) Les outils actuels peuvent-ils surveiller efficacement les systèmes distribués de grande taille, en particulier en présence de problèmes de synchronisation des traces et d'environnements hétérogènes ? (2) Comment optimiser les approches de traçage hybride afin de réduire le stockage, la surcharge et l'effort manuel, tout en fournissant une analyse précise des causes racines ? (3) Comment les modèles de langage de grande taille (LLMs) peuvent-ils aider à interpréter les journaux, les traces et les métriques système, tout en assurant l'évolutivité, la confidentialité et l'intégration avec d'autres méthodes de diagnostic de performance dans les systèmes distribués ?

1.1 Objectifs de recherche

Les systèmes distribués modernes alimentent une large gamme de services logiciels en exécutant plusieurs composants indépendants sur de nombreuses machines. Ces systèmes offrent une scalabilité élevée, de la flexibilité et une isolation des fautes, mais introduisent une complexité accrue dans la compréhension et le débogage des problèmes de performance. Lorsqu'une requête traverse de nombreux services — parfois de manière parallèle ou asynchrone — de petites inefficacités ou fautes peuvent se propager et devenir difficiles à retracer. Les outils de traçage de bout en bout traditionnels ne fournissent qu'une vue d'ensemble de haut niveau, tandis que les métriques système de bas niveau manquent souvent de sens contextuel, en l'absence de lien avec les traces. De plus, la taille croissante des logs de traces et les préoccupations liées à la confidentialité en production exigent des méthodes d'analyse qui soient à la fois scalables, précises et sécurisées. Cette recherche s'attaque au défi de diagnostiquer avec précision les problèmes de performance dans les systèmes logiciels distribués à grande échelle, où la complexité, le volume et l'hétérogénéité des données rendent l'analyse des causes profondes difficile. Le travail propose un ensemble de techniques intégrées combinant la comparaison avancée de traces, l'analyse des métriques basée sur le chemin critique, et l'interprétation des journaux à l'aide de méthodes statistiques et de modèles de langage de grande taille (LLMs). Ces approches visent à réduire le temps d'analyse manuel, à améliorer la précision et à permettre un diagnostic automatisé et évolutif en environnement industriel.

1. **Comparaison visuelle de traces distribuées à l'aide de graphes de flammes enrichis :** La première contribution porte sur la comparaison des performances entre différents groupes de requêtes dans des systèmes à base de microservices, à travers un outil appelé DTraComp. Ce travail répond à l'absence de fonctionnalités de comparaison visuelle dans les outils de traçage existants et introduit un graphe de flammes différentiel intégré à Eclipse Trace Compass. Il combine la visualisation d'intervalles (spans) de haut niveau avec des métriques d'état de thread au niveau système, permettant une analyse précise des causes racines à travers plusieurs machines. La méthode est testée sur des applications distribuées réelles et prend en charge à la fois les services séquentiels et parallèles. Une implémentation publique et une évaluation détaillée démontrent l'efficacité de l'approche pour identifier les sources de latence.
2. **Analyse légère des causes racines via un traçage hybride sensible au chemin critique :** La deuxième contribution aborde les limites des méthodes actuelles d'analyse de causes racines, qui nécessitent souvent un stockage important et souffrent d'un mauvais alignement des traces. Cette approche combine du traçage distribué de haut niveau avec une surveillance système de bas niveau, en collectant des métriques dé-

taillées uniquement pour les services présents sur le chemin critique. Cette méthode de traçage hybride sélectif réduit la surcharge tout en maintenant une grande précision. La technique proposée est validée sur plusieurs bancs d'essai à base de microservices, montrant une amélioration du rappel et de la précision par rapport aux méthodes existantes, tout en utilisant significativement moins de stockage et de ressources de calcul.

3. **Analyse des causes racines basée sur les logs à l'aide de LLMs optimisés pour la confidentialité :** La troisième contribution porte sur l'amélioration de la compréhension du comportement des systèmes en utilisant des modèles de langage de grande taille (LLMs) pour analyser les journaux (logs), les traces et les métriques système. Ce travail présente InsightAI, un système conçu pour aider les ingénieurs et les opérateurs en utilisant l'apprentissage en contexte, le découpage adaptatif, et une méthode de compression de jetons inspirée des flame graphs. Ces techniques permettent de réduire le nombre de jetons envoyés aux LLMs, diminuant ainsi les coûts et la latence tout en préservant le sens des données. En plus d'analyser les journaux, InsightAI permet aussi de comprendre les traces et d'interpréter les tendances des métriques, offrant une vision plus complète de l'état du système. Le système intègre également une nouvelle méthode pour anonymiser les données sensibles, essentielle dans les environnements industriels. Le système a été testé en collaboration avec des partenaires industriels, montrant des avantages concrets en termes de performance, de protection des données et de facilité d'utilisation.

1.2 Plan de thèse

Le reste de cette thèse est organisé comme suit. Le Chapitre 2 présente les concepts de base ainsi qu'une revue détaillée des travaux existants sur l'analyse de performance et le traçage dans les systèmes distribués. Le Chapitre 4 expose la première contribution de recherche, en introduisant DTraComp, un cadre pour la comparaison de traces distribuées à l'aide de visualisations enrichies et de métriques système. Le Chapitre 5 discute d'une méthode basée sur le traçage hybride pour une analyse légère des causes racines, centrée sur les problèmes de latence liés au chemin critique. Le Chapitre 6 présente un cadre d'analyse de logs utilisant des LLMs, conçu pour gérer des données privées et minimiser l'utilisation de jetons. Enfin, le Chapitre 7 résume les contributions de cette thèse, en identifie les limites, et propose des pistes pour des travaux futurs.

1.3 Publications

1. Maryam Ekhlasi, Anurag Prakash, Michel Dagenais, Maxime Lamothe, “InsightAI : Root Cause Analysis in Large Hierarchical Log Files with Private Data Using Large Language Models”, International Conference on AI Engineering - Software Engineering for AI (CAIN), 2025, accepted.
2. Maryam Ekhlasi, Fatemeh Faraji Daneshgar, Michel Dagenais, nasser Ezzati-Jivan, Matthew Khouzam, “DTraComp : Comparing distributed execution traces for understanding intermittent latency sources”, Journal of Systems and Software, 2025, under review.
3. Maryam Ekhlasi, Michel Dagenais, Naser Ezzati-Jivan, Maxime Lamothe, “Lightweight Root Cause Analysis for Latency Issues Using Critical Path-Aware Hybrid Tracing”, International Conference on Software Maintenance and Evolution (ICSME), 2025, under review.

CHAPITRE 2 REVUE DE LA LITTÉRATURE

2.1 Analyse des performances

Les systèmes distribués sont constitués de nombreux composants interconnectés, offrant des services à un très grand nombre d'utilisateurs potentiels, ce qui impose une pression significative sur les performances. Il est donc essentiel de mettre en place des mécanismes de surveillance et de collecte de données permettant d'obtenir des informations sur les performances du système en production. Cette tâche devient encore plus complexe lorsque ces applications sont déployées en infonuagique. Des mécanismes appropriés doivent être considérés et calibrés afin de fournir une compréhension complète de l'état d'un système ou d'une opération à un moment donné.

Pour répondre aux attentes des utilisateurs, les systèmes logiciels doivent exécuter leurs fonctions avec des performances et une fiabilité acceptables. Cependant, les performances de ces systèmes peuvent être affectées par des problèmes survenant à l'exécution, tels qu'une dégradation du temps de réponse [4]. Les anomalies de performance se traduisent par un comportement du système surveillé qui ne peut être expliqué par la charge de travail actuelle [5]. À titre d'exemple, lorsqu'un système traite un certain nombre de requêtes, la consommation de CPU et de mémoire peut être beaucoup plus élevée que ce qui serait attendu pour cette charge.

Pour maintenir les performances des systèmes logiciels, les opérateurs doivent identifier les anomalies de performance et prévenir les défaillances durant l'exécution [6]. À mesure que les systèmes deviennent plus vastes et complexes, il devient difficile pour les opérateurs de surveiller manuellement leur état d'exécution. C'est pourquoi des chercheurs ont proposé différentes méthodes pour surveiller automatiquement les systèmes logiciels et détecter les anomalies à l'exécution [7–11].

Parallèlement, il existe divers outils permettant d'évaluer les performances logicielles à un niveau élevé, comme au niveau utilisateur dans les applications monolithiques, ou via du traçage de bout en bout pour les systèmes distribués, ce que nous détaillons dans la Section 2.2. Pour la collecte de traces, nous examinons plusieurs outils, décrits plus en détail dans la Section 2.4. Étant donné que ce travail se concentre principalement sur les systèmes distribués, nous mettons l'accent sur les outils populaires qui suivent le modèle de "distributed open tracing" dans la Section 2.5. Enfin, la Section 2.7 présente les travaux antérieurs sur ce sujet, y compris les études portant sur l'analyse de performance des applications distribuées et le

traçage logiciel de bas niveau.

2.2 Définitions et terminologie

Cette section couvre l'état de l'art en lien avec notre proposition de thèse. Tout d'abord, nous présentons les termes généraux et les définitions de base liés à notre travail. Ensuite, nous décrivons les outils de traçage les plus importants et les plus populaires, en exposant leurs avantages et leurs limitations. Nous fournissons également des informations sur les outils standards du modèle "distributed open tracing", ainsi que sur certains outils de visualisation. Enfin, nous expliquons différentes techniques de mesure et d'analyse de performance, en couvrant diverses approches de détection d'anomalies de performance, avec leurs contributions et limitations.

2.2.1 Tracer

Le traçage est une méthode permettant de comprendre ce qui se passe lors de l'exécution d'un système logiciel. Un traceur désigne le composant logiciel utilisé pour effectuer le traçage. Il s'agit d'une technique permettant de comprendre le comportement d'une application en recueillant des données sur le chemin d'exécution de l'application instrumentée. Le traçage logiciel est largement utilisé pour inspecter les systèmes pendant leur exécution, afin d'identifier les problèmes de performance et d'en exposer les causes profondes [12]. Contrairement au logging, le traçage enregistre généralement des millions d'événements de bas niveau à haute fréquence, souvent avec un volume élevé. Les traces peuvent contenir des événements provenant du noyau du système d'exploitation, tels que l'activité de planification, l'activité réseau, l'entrée et la sortie des gestionnaires d'interruptions (IRQ), l'entrée et la sortie des appels système, etc. Elles peuvent également contenir des événements provenant des applications en espace utilisateur, aussi appelés User Space Tracing (UST), ou une combinaison d'événements de niveau utilisateur et système.

— **User-space tracing :** Pour obtenir des événements en espace utilisateur, il est nécessaire d'ajouter des points de trace au code source, soit via une instrumentation statique (avant la compilation), soit via une instrumentation dynamique (après la compilation) directement dans l'exécutable binaire. L'instrumentation statique est plus simple à mettre en œuvre que l'instrumentation dynamique, mais elle nécessite l'accès au code source ainsi qu'une recompilation. LTTng-UST est un outil de la suite LTTng qui inclut un agent capable de tracer également du code Java et Python.
— **Kernel-space tracing :** Le tracing en espace utilisateur dépend fortement du code

applicatif et varie d'une application à l'autre. Il ne permet pas de capturer des événements au niveau du système d'exploitation, comme les réveils de threads, le verrouillage de fichiers ou les événements de l'ordonnanceur. En revanche, le traçage en espace noyau (kernel-space) utilise des points de trace instrumentés dans le noyau du système d'exploitation, indépendamment de l'application. Avec ce type de traçage, les développeurs n'ont pas besoin de modifier leur code pour ajouter des points de trace, évitant ainsi tout impact direct sur celui-ci. De plus, le traçage en espace noyau capture des événements provenant de l'ensemble du système, y compris de l'application et du système d'exploitation. Cela est particulièrement utile pour détecter un comportement anormal pouvant être externe à l'application. Il est possible de collecter des traces issues de l'espace utilisateur, de l'espace noyau, ou des deux simultanément.

Il convient de mentionner que tracer une application impose une surcharge supplémentaire sur les performances du système, dans le but d'obtenir des informations précieuses permettant d'identifier les causes des problèmes de performance, tels que les ralentissements ou les pannes du système. Cette surcharge supplémentaire, appelée coût du traçage, doit être acceptée par les développeurs afin d'obtenir une vue en temps réel du comportement du système. Il est donc impératif de sélectionner un outil adapté, avec une surcharge minimale, pour évaluer les performances logicielles. Cela garantit que le comportement du système étudié est aussi peu perturbé que possible par le processus de traçage lui-même. En Section 2.4, nous décrivons les outils de traçage les plus populaires, en précisant leur impact sur le système.

2.2.2 Débogueur

Un débogueur est un outil qui permet aux programmeurs de surveiller l'exécution d'un programme et d'identifier tout problème pouvant entraîner un dysfonctionnement. Il fonctionne en exécutant le programme dans des conditions contrôlées et offre différentes fonctionnalités, telles que la possibilité d'arrêter le programme à des points spécifiques. Il permet également d'afficher le contenu de la mémoire et des registres, ainsi que de modifier les valeurs en mémoire afin de tester le comportement du programme dans différentes conditions [13].

Le débogage et les tests sont deux processus complémentaires visant à améliorer la qualité des logiciels. Tandis que les tests ont pour objectif d'identifier des erreurs dans le code source d'un programme, le débogage se concentre sur la localisation et la correction de ces erreurs. Les tests ne permettent pas de localiser précisément l'erreur dans le code, mais ils révèlent son impact sur le comportement du programme. Une fois l'erreur identifiée, le débogage devient crucial pour en déterminer la cause racine, afin de la corriger efficacement [14].

Les débogueurs sont particulièrement utiles pour identifier et résoudre les problèmes à l'exé-

cution, tels que les plantages, les fuites de mémoire ou les comportements atypiques. Ils permettent aux programmeurs d'examiner leur code ligne par ligne et d'observer l'état du programme à différents moments, facilitant ainsi la localisation des causes des dysfonctionnements.

Il existe des débogueurs tiers qui peuvent être utilisés avec plusieurs langages de programmation, en plus des outils de débogage fournis avec la plupart des environnements de développement. Les débogueurs peuvent être utilisés depuis un environnement de développement intégré (IDE) ou via la ligne de commande.

Le traçage et le débogage sont deux outils essentiels du développement logiciel, mais ils ont des finalités différentes. Le choix entre traçage et débogage dépend de l'objectif visé par le développeur. Si l'objectif est de trouver et corriger une erreur dans le code, le débogage est l'outil approprié. En revanche, si l'objectif est de surveiller les performances d'un programme et d'identifier les pistes d'amélioration, le traçage est plus adapté.

2.2.3 Profileurs

Le profilage est une technique qui permet de suivre des statistiques de performance lorsqu'un événement spécifique se produit pendant l'exécution d'un programme. Ces statistiques sont mises à jour en temps réel et stockées dans un fichier de profil une fois le programme terminé. Elles résument les métriques de performance du programme et peuvent être analysées pour fournir à l'utilisateur un aperçu de son comportement [15].

Les profileurs sont des outils permettant d'identifier les parties les plus consommatrices en ressources dans un programme. Ils peuvent être utilisés pour localiser les goulets d'étranglement dans une portion de code en extrayant certaines métriques statistiques. Ces métriques peuvent inclure le nombre d'appels de fonctions, le temps CPU, ou le taux de succès en cache, entre autres. Toutefois, ces métriques ne permettent pas nécessairement d'identifier l'origine exacte du goulet d'étranglement. Ces derniers peuvent n'apparaître que dans certaines conditions, difficiles à détecter pour un profileur statistique, voire impossibles à reproduire.

Le profilage est une technique de haut niveau qui permet d'identifier des tendances générales et des motifs dans les performances d'un programme. Le traçage de bas niveau, en revanche, est une technique plus fine et plus détaillée, utilisée pour diagnostiquer des problèmes spécifiques dans un programme. Par exemple, une latence importante hors CPU, pouvant survenir lors de l'attente d'un accès au processeur, de la lecture d'un fichier ou de la réception d'un paquet réseau, est généralement le signe d'une surcharge du système ou d'un problème matériel. Dans ce type de situation, les profileurs ne sont pas particulièrement efficaces pour identifier

la cause sous-jacente. Ils se contentent de classer les fonctions ou les modules en fonction de métriques statistiques, sans fournir d'informations sur la nature des problèmes [16].

2.3 traçage de bout en bout

Le traçage de bout en bout (end-to-end tracing) est activé dans l'ensemble d'un système distribué afin de faciliter la compréhension et le diagnostic des problèmes potentiellement causés par les dépendances complexes entre services dans les architectures à base de microservices. Il permet d'enregistrer à la fois le comportement des composants individuels du système et leurs relations. Une manière de suivre les requêtes consiste à maintenir un contexte global de requête (request ID) à travers les composants logiciels et les machines, pendant le traitement d'une requête [17].

Contrairement à la collecte de métriques ou au logging, le traçage distribué offre une vue unifiée de la propagation des requêtes à travers les frontières des composants dans un système distribué [18]. Le traçage de bout en bout se caractérise par les éléments suivants [19] :

— Des événements de trace peu coûteux et de granularité fine, avec des horodatages haute précision, capturant l'utilisation des ressources comme les entrées/sorties disque, les allocations de mémoire tampon ou les changements de contexte.
— Le traçage ne s'effectue pas uniquement sur les chemins anormaux, mais aussi sur les chemins rapides ou fréquents.
— Une comptabilisation précise des ressources est rendue possible grâce à la capture de chaque instance d'utilisation de ressource et de changement de contexte, ce qui permet de distinguer la demande imposée au système et les services qu'il fournit.
— La séparation entre demande et service est assurée par le suivi du contexte d'exécution de chaque événement d'utilisation de ressource. Cela permet de différencier l'utilisation des ressources par chaque requête et la planification de ces requêtes sur les threads et processeurs sous-jacents. Ce point est particulièrement important dans les systèmes hautement concurrents, qui multiplexent de nombreuses requêtes sur un nombre réduit de contextes d'exécution, tels que les threads et processeurs.
— La synchronisation et le transfert de contrôle sont tracés en utilisant des contextes globaux, comme les identifiants de requête, ce qui permet de définir une requête à différents niveaux de granularité. Cette approche est également utile pour diagnostiquer les goulets d'étranglement liés à la synchronisation.
— En plus de la concurrence entre requêtes indépendantes, une requête peut présenter de la concurrence interne, due au traitement parallèle, aux entrées/sorties asynchrones, ou aux appels RPC asynchrones. Il est possible d'extraire cette concurrence comme

un ordre partiel en suivant les changements de contexte et les événements de synchronisation.

Le traçage de bout en bout présente plusieurs avantages par rapport aux compteurs de performance, notamment :

— Une vue désagrégée des requêtes individuelles, offrant une compréhension détaillée des performances et de l'utilisation des ressources pour chaque requête.
— Des informations sur le flux de contrôle, incluant l'ordonnancement, la concurrence et les dépendances internes à une requête.
— Une comptabilisation précise de chaque événement d'utilisation de ressource, assigné à une requête unique.
— Une grande flexibilité pour ajouter de nouveaux compteurs, en calculant différents agrégats à partir des données de trace.
— L'analyse du temps de réponse, qui n'est pas possible avec les compteurs de performance seuls. Le traçage de bout en bout permet de suivre l'utilisation des ressources par requête ainsi que les dépendances d'ordonnancement, permettant ainsi la mesure et la prédiction de la latence d'exécution.

2.3.1 Système d'états

Comme illustré dans la Figure 2.1, nous avons un cycle de vie des processus dans lequel chacun passe par différents états. Un processus peut avoir plusieurs états, tels que : En cours d'exécution (Running), Prêt à s'exécuter (exécutable mais préempté), Né ou dupliqué, En sommeil (Sleeping), Sommeil non interruptible (Uninterruptible sleep), et État zombie (Zombie).

L'appel système fork est utilisé pour créer des processus, et une fois démarré avec succès, POSIX crée deux copies identiques des espaces d'adresses, pour le processus parent et le processus enfant. Ensuite, les processus parent et enfant passent à l'état en cours d'exécution ou prêt à s'exécuter. Lorsqu'un processus est exécuté sur un cœur de processeur, le CPU est dans l'état d'exécution, qui peut être en mode utilisateur ou en mode noyau.

Prêt à s'exécuter (runnable) signifie que le processus a besoin de ressources (autres que le CPU) pour fonctionner, telles que des opérations d'E/S disque ou réseau, qui ont lieu dans l'espace noyau. Bloqué, en attente ou en sommeil (en mémoire ou sur disque), en sommeil interruptible ou non interruptible, signifie que le processus n'a pas les ressources requises ou

attend un événement spécifique ou un créneau horaire. Les états terminé ou arrêté indiquent que le processus a terminé son exécution ou a reçu un signal d'arrêt (par exemple, un signal SIGCHLD envoyé à son parent). La période pendant laquelle un processus enfant attend que son parent libère son créneau est appelée l'état Zombie.

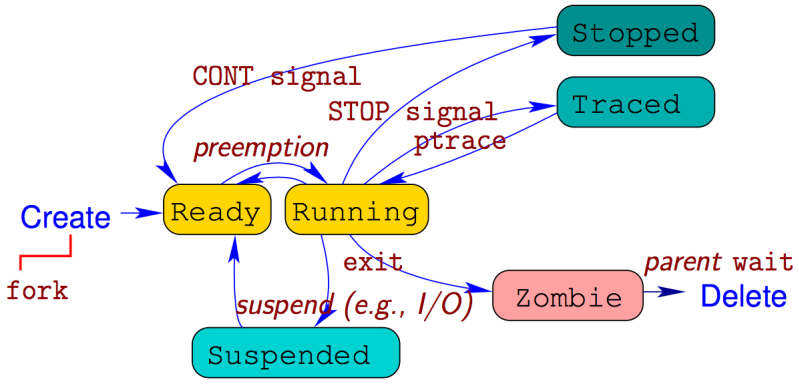


FIGURE 2.1 L'état système d'un processus

2.3.2 Métriques système

Nous disposons de différentes métriques système, incluant l'utilisation du CPU, les opérations d'E/S (comme le réseau et les périphériques de bloc), les allocations mémoire et les opérations sur fichiers.

— L'utilisation du CPU peut être dans deux états : inactif (Idle) ou ordonnancé (Scheduled). Lorsqu'aucun processus ne s'exécute sur un CPU, celui-ci est en mode inactif et le noyau exécute le thread Swapper. Sinon, il est en mode ordonnancé, et le temps CPU peut être calculé en ajoutant des intervalles au temps pendant lequel un processus est exécuté.

— Les opérations sur fichiers sont calculées à l'aide des appels système open, read, write et close.

— Concernant l'utilisation mémoire, lorsqu'une page mémoire est allouée ou libérée, les événements du noyau page alloc et page free sont générés. Les opérations réseau sur les sockets sont similaires aux fichiers, où les événements net.socket et net.accept sont liés à la création de sockets, et net.socket_dengmsg et net.socket_recvmsg sont les événements de transfert.

2.4 Outils de traçage

Les outils de traçage sont des utilitaires logiciels permettant aux développeurs et aux administrateurs système de surveiller et d'analyser le comportement des applications logicielles et des systèmes d'exploitation. Ces outils fonctionnent en ajoutant des instructions supplémentaires dans le code logiciel pour capturer divers événements, tels que les appels système et les changements de variables. Les données ainsi collectées sont ensuite utilisées pour offrir une vue complète du fonctionnement du programme durant son exécution. Ci-dessous sont présentés les outils les plus importants, accompagnés de leurs fonctionnalités et limitations.

2.4.1 LTTng

LTTng, le Linux Trace Toolkit next generation, est un outil de traçage en code source libre, à faible surcharge et haute performance, permettant une analyse intégrée à la fois de l'espace noyau et de l'espace utilisateur. Il utilise environ 2 % du temps CPU sous une charge de travail élevée.

Il permet l'instrumentation de points de trace statiques, et supporte également l'instrumentation de points de trace dynamiques via les Linux Kernel Markers et les Kprobes.

LTTng est composé de trois parties principales : lttctl, lttd, et une partie noyau. Lttctl est une application en ligne de commande en espace utilisateur permettant de contrôler les traces en lançant lttd. Lttd est un démon en espace utilisateur qui écrit les données de trace sur disque ou les exporte en utilisant le format CTF (Common Trace Format) [20]. Les développeurs, testeurs et administrateurs système peuvent utiliser les points de trace statiques dans leur code source et tirer parti de LTTng pour enregistrer des informations d'exécution sur leurs applications en temps réel. Pour compléter les points de trace statiques, les développeurs peuvent également insérer dynamiquement de nouveaux points de trace via un appel système, une interruption logicielle (trap), ou un saut surimposé. Les points de trace dynamiques peuvent être ajoutés au noyau Linux via kprobe, ou aux applications en espace utilisateur via les points de trace du débogueur GNU (GDB) [21] ou via uprobe. Les composants principaux de LTTng et leurs interactions sont illustrés à la Figure 2.2.

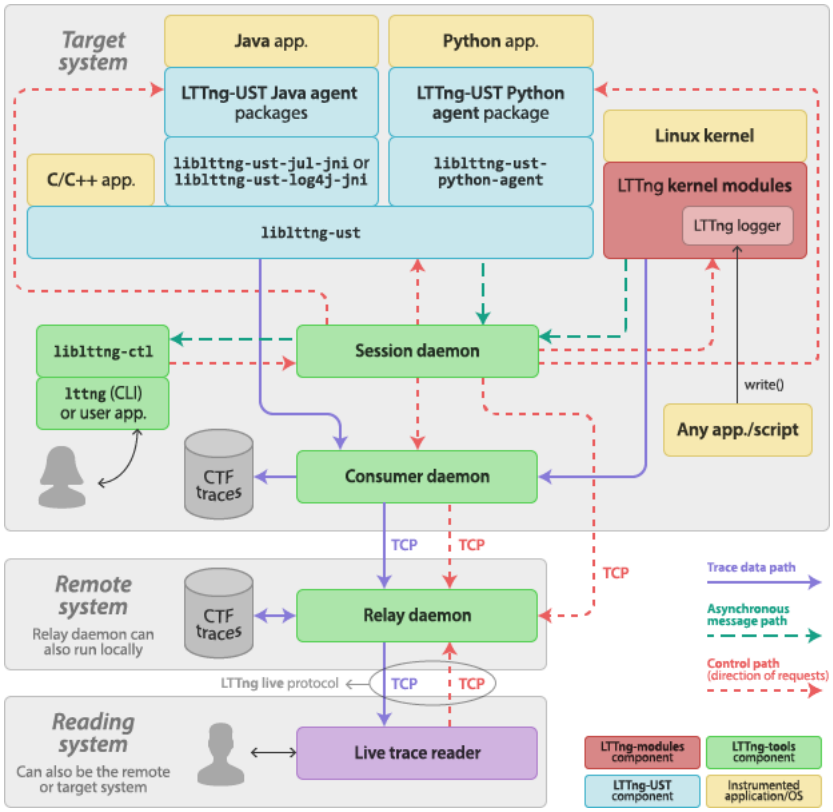


FIGURE 2.2 Chemins de contrôle et de données de trace entre les composants LTTng

2.4.2 FTrace

FTrace, ou Function Tracer, est un autre outil de Linux qui fait désormais partie du noyau Linux et est activé par défaut. Il utilise des points de trace statiques dans le noyau. Il s'appuie essentiellement sur le cadriciel Frysk pour tracer les appels système. Contrairement à LTTng, il s'adresse uniquement aux développeurs du noyau, car il se concentre sur les traces du noyau, telles que celles de l'ordonnanceur, des pilotes, de la gestion mémoire et de la couche bloc. Il ne permet pas la collecte de traces en espace utilisateur [20]. FTrace peut générer une quantité considérable de données de trace, ce qui peut entraîner une dégradation des performances du système, en raison de la surcharge engendrée. En comparaison, LTTng peut également produire un volume important de données, mais il offre un contrôle plus précis du traçage, permettant d'ajuster la configuration pour minimiser l'impact sur les performances du système.

2.4.3 eBPF

eBPF signifie extended Berkeley Packet Filter. Il s'agit d'une machine virtuelle (VM) et d'un cadriciel permettant d'exécuter du code personnalisé à l'intérieur du noyau Linux. eBPF permet aux développeurs d'ajouter des fonctionnalités au noyau sans le modifier, ce qui peut améliorer la sécurité, la stabilité et les performances. Les programmes eBPF s'exécutent lorsqu'un événement spécifique survient dans le noyau, comme la réception d'un paquet réseau ou l'appel à une fonction système. Les programmes sont écrits dans un sous-ensemble restreint du langage C et compilés en bytecode, qui est ensuite chargé dans le noyau à l'exécution. Ce bytecode est vérifié pour sa sûreté et sa sécurité avant d'être exécuté, ce qui empêche les plantages ou les vulnérabilités de sécurité [22]. L'architecture générale de eBPF est illustrée à la Figure 2.3.

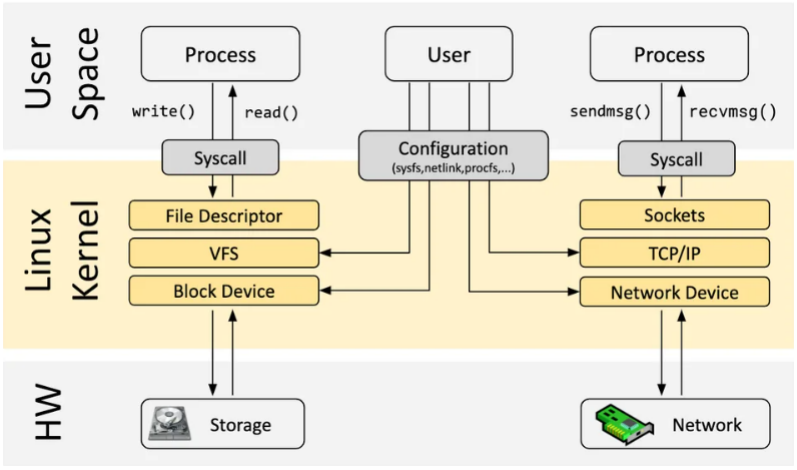


FIGURE 2.3 Architecture de eBPF

eBPF est conçu pour avoir une faible surcharge, mais nécessite tout de même des ressources pour compiler, charger et exécuter les programmes eBPF. Dans certains cas, en particulier sur des systèmes à ressources limitées, cette surcharge peut être suffisante pour impacter les performances ou la disponibilité des ressources. Bien que eBPF soit un cadriciel puissant et flexible pour le traçage sur les systèmes Linux, il présente aussi certaines limites. Il est principalement conçu pour le traçage au niveau noyau, ce qui signifie qu'il n'est pas toujours adapté pour des applications nécessitant une analyse détaillée des événements et métriques en espace utilisateur. Il reste cependant un outil à faible surcharge, bien que cette surcharge puisse devenir significative dans certains environnements.

2.4.4 Perf

Perf est un outil d'analyse de performance et de profilage pour les systèmes Linux. Il s'agit d'un outil en ligne de commande offrant une gamme étendue de fonctionnalités pour analyser les performances des systèmes et des applications. Il peut instrumenter les compteurs de performance CPU, les points de trace, les kprobes et les uprobes (traçage dynamique). Bien qu'il s'agisse d'un outil flexible pour l'analyse et l'optimisation des performances, son utilisation peut s'avérer complexe. Il requiert une compréhension approfondie de l'architecture du système et des métriques de performance, ainsi qu'une expertise dans les techniques de profilage et d'optimisation [23].

Perf est un bon choix lorsque les utilisateurs souhaitent analyser l'utilisation du CPU ou de la mémoire, ou s'ils ont besoin d'outils spécialisés pour une analyse détaillée des données de performance. Cependant, LTTng est une meilleure option si l'on recherche un outil avec une surcharge minimale offrant un traçage à faible latence.

2.4.5 SystemTap

SystemTap (System Tool for Analysis Program) est un outil de traçage et d'investigation pour systèmes Linux. Il est en code source libre et bénéficie de contributions de Red Hat, IBM, Intel, Hitachi, Oracle, l'Université du Wisconsin-Madison, ainsi que d'autres membres de la communauté. Il permet d'observer et d'analyser les activités du système en temps réel, sans modifier le noyau ni redémarrer le système.

SystemTap offre un langage de script puissant et flexible permettant de réaliser des tâches complexes de traçage et d'analyse. Les scripts sont écrits dans un langage proche du C et peuvent accéder à un ensemble riche de structures de données système et de fonctions. Il permet de définir des sondes personnalisées pour tracer des événements spécifiques du système, tels que les appels système, les fonctions du noyau et les fonctions en espace utilisateur, afin de surveiller les performances du système, analyser l'exécution du code et diagnostiquer des problèmes système [24, 25].

SystemTap n'a pas été intégré au noyau Linux principal. Beaucoup considèrent que eBPF est désormais l'outil officiel de traçage dans cette catégorie. Bien que SystemTap soit un outil puissant pour le traçage et l'investigation des systèmes Linux, il peut ne pas convenir aux utilisateurs ayant besoin d'un traçage à faible surcharge, d'un support multiplateforme ou d'un modèle de script simplifié.

2.5 Outils de traçage distribués

Plusieurs logiciels de traçage de bout en bout ont été développés, chacun avec des caractéristiques spécifiques, qui sont décrites en détail dans les sous-sections suivantes.

2.5.1 OpenCensus

OpenCensus [26] est un ensemble à code source libre de bibliothèques permettant la collecte de métriques applicatives et de traces à travers un système distribué. Il a été initialement développé par Google, à partir du projet libre Census, dans le but de capturer des traces et des métriques entre plusieurs fournisseurs de services infonuagiques. Il prend en charge de nombreux langages de programmation et peut être utilisé avec divers fournisseurs cloud et plateformes de supervision. Grâce à OpenCensus, les développeurs peuvent instrumenter leurs applications via une API unique et envoyer des données de télémétrie vers différents collecteurs, tels que Stackdriver, Prometheus et Jaeger. Le projet a fusionné avec OpenTelemetry en 2019 afin de créer un standard unifié pour les systèmes distribués modernes.

2.5.2 OpenTracing

OpenTracing [27], tout comme OpenCensus, est un ensemble d'API et de bibliothèques en code source libre, indépendantes des fournisseurs, visant à standardiser l'instrumentation du code dans les systèmes distribués pour la collecte de données de télémétrie. Il a été développé principalement par la société Uber. OpenTracing est compatible avec le projet Jaeger, et peut stocker les données de trace dans deux types de bases : Cassandra et ElasticSearch. Jaeger, pour sa part, est un outil de traçage distribué de bout en bout doté d'une interface utilisateur minimale pour l'analyse des données capturées.

2.5.3 OpenTelemetry

De nombreux outils de traçage utilisent leur propre processus d'instrumentation, ce qui nécessite un effort distinct pour chaque service. De plus, la compatibilité entre les outils de différents fournisseurs n'est pas toujours assurée, ce qui peut entraîner une perte de contexte lors de la résolution des problèmes, car certains services peuvent ne pas être compatibles avec tous les outils [28]. OpenTelemetry a réuni les deux technologies les plus matures, OpenTracing et OpenCensus. Ce projet est largement reconnu par les communautés de l'infnuagique et des logiciels libres. Son objectif principal est de généraliser et de standardiser le traçage dans les applications infonuagiques, avec pour but de fournir une source fiable de données de trace via l'instrumentation.

OpenTelemetry reprend plusieurs concepts issus de OpenTracing, notamment celui d'intervalle (Span), qui décrit la durée d'une action à un instant donné. Un Span est associé à plusieurs attributs permettant de fournir une description complète de l'action mesurée. Un autre concept introduit par OpenTelemetry est celui de "resource", qui identifie l'entité réalisant la mesure et est accompagné d'attributs décrivant le processus par lequel la mesure est effectuée [18].

Étant donné que notre application ne s'exécute généralement pas sur la machine étudiée, nous avons besoin d'une vue centrée sur la requête dans l'environnement microservices [29]. Plusieurs standards existent dans les systèmes à base de microservices (par exemple : OpenTracing, OpenCensus et OpenTelemetry). En effet, OpenTracing et OpenCensus étaient deux standards concurrents. Toutefois, OpenTelemetry fournit une solution complète au problème du traçage de bout en bout dans les systèmes à base de microservices, en combinant les aspects des deux précédents standards dans un cadre unifié.

2.6 Outils de visualisation des traces

Les événements de traçage peuvent être observés sous forme de fichiers journaux afin d'obtenir un niveau de détail maximal. Néanmoins, il est possible d'extraire des informations utiles à partir des données brutes en utilisant des analyseurs et visualiseurs de traces, associés à une bonne compréhension du programme tracé. Ces outils doivent être capables de traiter le volume massif de données que peut contenir une trace.

Par exemple, une vue du flux de contrôle (Control Flow View) dans Trace Compass est illustrée à la Figure 2.4. Elle montre la relation parent-enfant entre processus, et les barres colorées représentent les états du processus correspondant dans l'arborescence.

2.6.1 Approches de visualisation

Graphe de flammes

Un graphe de flammes, illustré à la Figure 2.5, est un outil de visualisation utilisé pour profiler les performances logicielles. Brendan Gregg l'a introduit pour la première fois en 2011. Un graphe de flammes affiche la pile d'appels d'un programme ainsi que le temps passé dans chaque fonction. La pile d'appels est représentée sous forme de barres, chacune correspondant à une fonction. La profondeur de la pile est représentée sur l'axe vertical, du bas (racine) vers le haut (feuille). Le bord supérieur montre ce qui est en cours d'exécution

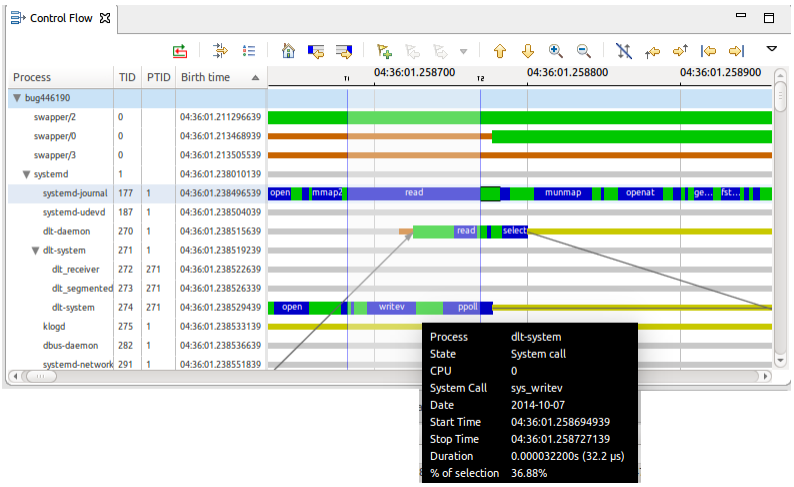


FIGURE 2.4 Vue du flux de contrôle dans Trace Compass

sur le CPU, et les éléments en dessous représentent son héritage.

L'axe horizontal représente les noms de fonctions, mais ce n'est pas une chronologie : l'ordre de gauche à droite n'a pas de signification temporelle. L'ordre est basé sur un tri alphabétique des noms de fonctions, depuis la racine de chaque pile jusqu'à la feuille. Cette méthode permet de maximiser la fusion des boîtes représentant les fonctions identiques. La largeur de chaque boîte indique la fréquence d'apparition de cette fonction dans les piles d'appels, ou comme ancêtre d'une fonction. Plus la boîte est large, plus la fonction est présente ou consomme du temps CPU. La couleur de fond de chaque boîte n'a pas de signification particulière et est choisie de façon aléatoire parmi des teintes chaudes [30].

Le nom graphe de flammes provient de son apparence, qui ressemble à une série de flammes s'élevant depuis une base. Il peut également être utilisé pour comparer les performances de différentes versions d'un même logiciel, ou pour comparer des configurations matérielles. Pour générer un graphe de flammes, il faut généralement utiliser un outil de profilage capable de produire une pile d'appels.

Flame Chart

Un diagramme de flammes est un outil de visualisation inspiré du graphe de flammes, qui affiche cette fois la progression temporelle de l'exécution d'un programme. Il a été introduit pour la première fois par le Web Inspector de Google Chrome (WebKit).

La principale différence entre un graphe de flammed et un diagramme de flammes réside dans l'axe des abscisses : alors que le graphe de flammes utilise un ordre alphabétique pour

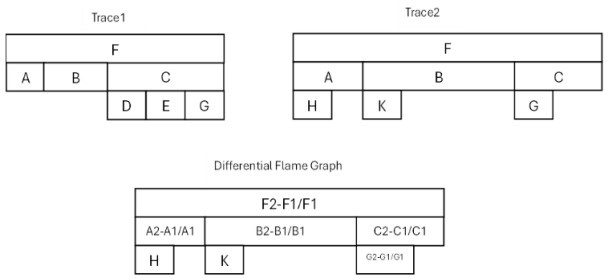


FIGURE 2.5 Aperçu d'un graphe de flammes

maximiser la fusion des fonctions, le diagramme de flammes représente l'évolution dans le temps. Cela permet d'examiner et d'analyser les motifs temporels dans l'exécution.

Le graphe de flammes permet une visualisation claire des programmes multithreadés, tandis que le diagramme de flammes, à l'origine conçu pour des applications JavaScript mono-threadées, n'offre pas une représentation significative dans un contexte multithread. Trace Compass propose les deux types de visualisations.

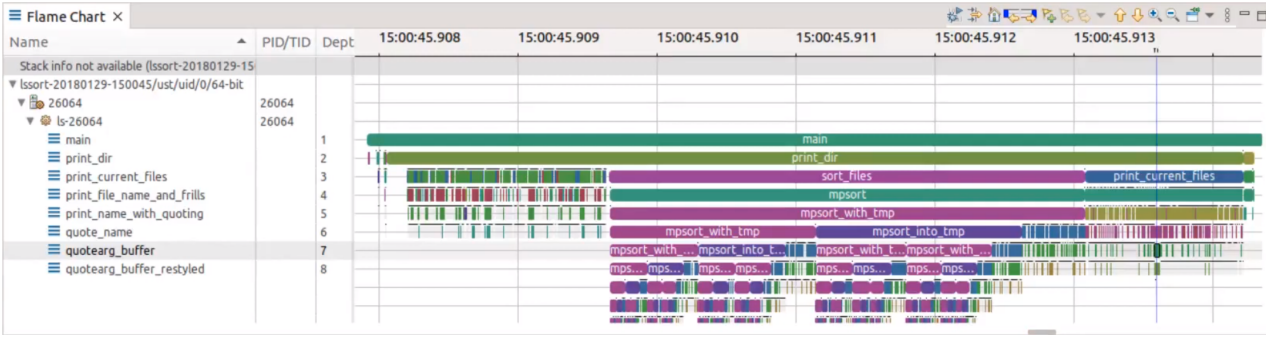


FIGURE 2.6 Aperçu d'un diagramme de flammes dans Trace Compass

Graphe de flammes Différentiel

Le graphe de flammes différentiel [31], illustré à la Figure 2.7, met en évidence les différences entre deux profils CPU distincts. Il représente visuellement les piles d'appels sur l'axe vertical, tandis que la taille et la forme des flammes indiquent les données du second profil (actuel ou postérieur).

La couleur de chaque rectangle indique le contraste entre le temps passé dans chaque pile d'appels dans les deux profils : le rouge indique une augmentation de charge, tandis que le bleu indique une diminution. Cette visualisation est utile pour comparer les performances de deux exécutions, mais elle est limitée à l'analyse de la consommation CPU.

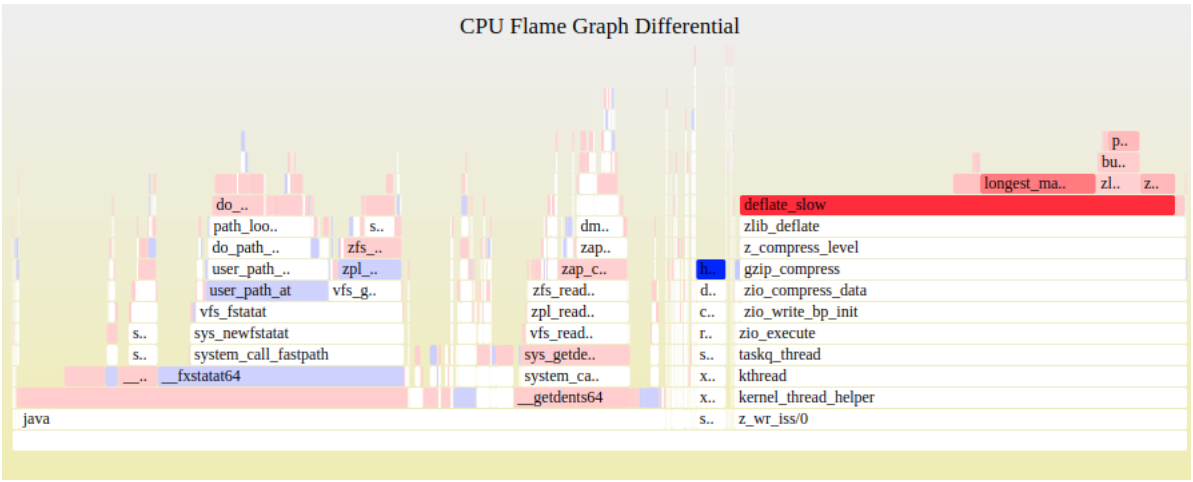


FIGURE 2.7 Aperçu d'un graphe de flammes différentiel

IntroPerf [32] est un profileur de performance fonctionnant sous Windows. Il permet d'inférer la performance à partir des piles d'appels obtenues par Event Tracing for Windows (ETW). Ce profileur compare le temps passé dans différentes fonctions sur la pile sur plusieurs exécutions. Contrairement au graphe de flammes différentiel, il prend également en compte les périodes d'attente hors CPU. Il affiche les différences entre exécutions via une carte thermique, chaque ligne représentant une pile d'appels unique, et chaque colonne représentant un niveau de la pile. Toutefois, IntroPerf n'est pas adapté à l'analyse d'applications multithreadées.

TraceCompare, illustré à la Figure 2.8 [33], introduit une nouvelle structure de données appelée Enhanced Calling Context Tree (ECCT), qui permet de représenter toutes les latences impactant la durée d'exécution des tâches. Cette structure permet d'identifier les dépendances entre threads et les temps d'attente hors CPU. L'approche repose sur le chemin critique : elle extrait les segments d'exécution avec leur état de thread, un horodatage de début et de fin. Un ECCT est construit pour ce chemin critique, puis un graphe de flammes différentiel enrichi est utilisé pour la comparaison.

Toutefois, leur méthode est limitée aux exécutions identiques et ne s'adapte pas bien aux changements majeurs du code. De plus, ils ne prennent pas en charge les bibliothèques standard de synchronisation en espace utilisateur, ce qui empêche l'extraction de certaines métriques importantes, notamment dans le contexte du traçage de bout en bout dans les systèmes distribués.

La proposition présentée dans le chapitre suivant vise à surmonter ces limitations : elle élargit le graphe de flammes différentiel pour inclure d'autres métriques que la consommation CPU, notamment les latences causées par les threads en attente. La visualisation, illustrée

à la Figure 2.10, comprend des paramètres tels que le temps d'exécution ou les octets lus depuis le disque, permettant de filtrer les exécutions à comparer. La Figure 2.9 montre comment les différences sont représentées par des couleurs (par exemple, rouge pour signaler une augmentation de latence), facilitant ainsi l'analyse visuelle.

FIGURE 2.8 TraceCompare : graphe de flammes différentiel enrichi

2.6.2 Applications de visualisation

Trace Compass

Eclipse Trace Compass est un logiciel open source conçu pour répondre aux problématiques de performance et de fiabilité en analysant et en interprétant des journaux de traces. L'objectif principal de cet outil est de proposer différentes visualisations, graphes, métriques et autres fonctionnalités facilitant l'extraction d'informations utiles à partir de traces, contrairement aux sorties textuelles souvent difficiles à lire.

Trace Compass offre un large éventail de fonctionnalités permettant de gérer efficacement des traces volumineuses, ainsi que des ensembles de traces. Ces fonctionnalités incluent notam-

FIGURE 2.9 Couleurs dans TraceCompare. Reproduit de [1]

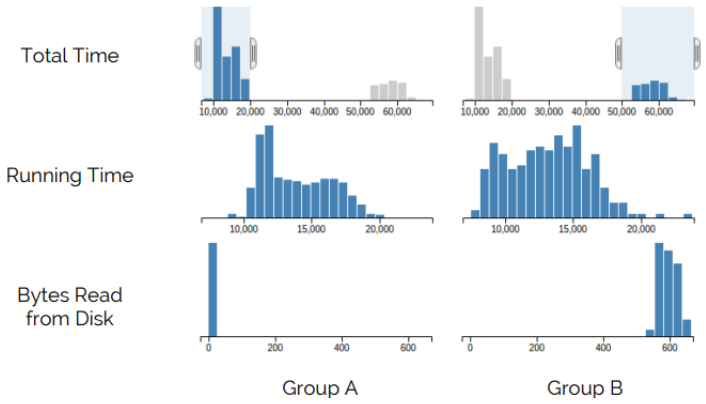


FIGURE 2.10 Filtres dans TraceCompare. Reproduit de [1]

ment la capacité à traiter des traces plus grandes que la mémoire disponible, la corrélation de multiples traces ordonnées temporellement, et un zoom jusqu'au niveau de la nanoseconde sur un segment donné d'une ou plusieurs traces. Il permet également la synchronisation des vues sur le temps sélectionné ou sur une plage temporelle donnée, la recherche et le filtrage efficaces d'événements, ainsi que la gestion de favoris de traces. De plus, Trace Compass permet l'importation et l'exportation de paquets de traces.

Trace Compass peut manipuler divers types de données de trace, notamment UFTrace, LTTng-UST, OpenTracing, CTF, Perf, PCAP, ainsi que d'autres formats. Il est également compatible avec des fichiers XML personnalisés [34].

Autres outils

Eclipse Theia est une plateforme destinée aux développeurs pour écrire, construire et déployer des applications en utilisant divers langages de programmation. Il s'agit d'un environnement open source, léger et flexible, avec une architecture modulaire qui permet aux utilisateurs de personnaliser leur IDE selon leurs besoins. Theia propose un ensemble de fonctionnalités

pour l'édition de code, le débogage, les tests et le contrôle de version, et peut être utilisé aussi bien sur un poste de travail que dans un navigateur.

Theia est construit avec des technologies web modernes telles que TypeScript, HTML et CSS, ce qui le rend léger et facilement adaptable. Son architecture modulaire permet d'ajouter ou de retirer des extensions en fonction des exigences spécifiques de chaque équipe ou organisation.

L'un des avantages majeurs de Theia est sa capacité à fonctionner à la fois dans un navigateur et sur un poste local, ce qui en fait une solution idéale pour les environnements de développement en infonuagique. De plus, Theia est hautement personnalisable et peut être facilement étendu pour répondre aux besoins particuliers de différentes équipes de développement [35].

Trace Compass est spécifiquement conçu pour le traçage et l'analyse, tandis que Theia est un environnement de développement généraliste. Cela signifie que Trace Compass fournit des fonctionnalités spécialisées pour le traitement des traces, avec un support étendu pour de nombreux formats de données de traçage. En revanche, Theia offre un environnement de développement flexible et personnalisable, basé sur l'infonuagique.

Ces dernières années, Trace Compass a été réarchitecturé pour séparer l'analyse en arrière plan et l'interface utilisateur en avant plan, via un protocole de communication appelé Trace Server Protocol (TSP). Une nouvelle extension Theia, appelée theia-trace-extension, a été développée pour prendre en charge ce protocole. Il est donc désormais possible d'utiliser Trace Compass soit avec l'interface Eclipse classique, soit avec l'interface Theia.

2.7 Approches de détection d'anomalies de performance

Les contentions, saturations, interblocages (deadlocks) ou défaillances système peuvent affecter les performances d'un système [36]. Cependant, de nombreux facteurs peuvent être à l'origine de ces problèmes : une mauvaise configuration, un réglage sous-optimal du système, des mises à jour applicatives, du code défectueux, des événements transitoires, ou encore des reconfigurations de la plateforme peuvent également entraîner des anomalies de performance.

La saturation des ressources correspond à une situation où une ressource atteint sa limite, provoquant des délais qui affectent la latence des requêtes. Bien que ces limitations varient selon le type de service, elles peuvent concerner l'utilisation du CPU, de la mémoire ou la mise en file d'attente de requêtes disque. Plusieurs méthodes existent pour gérer cette saturation, telles que l'utilisation de files de messages pour absorber la pression, ou le redimensionnement dynamique des ressources.

La contention de ressources se produit lorsque plusieurs processus accèdent simultanément à une même ressource, engendrant une latence significative. Ce phénomène est souvent

difficile à diagnostiquer, car il ne génère pas directement de métriques visibles [18].

Les méthodes d'analyse de performance les plus largement applicables ne nécessitent pas de modification du code source de l'application. Certaines exploitent des techniques d'apprentissage automatique pour déduire la configuration optimale, tandis que d'autres reposent sur l'injection contrôlée de fautes afin d'évaluer les performances dans des scénarios extrêmes [37, 38].

2.7.1 Approches statistiques et travaux basés sur le traçage

Au fil des années, de nombreux travaux ont été réalisés pour détecter et diagnostiquer les anomalies de performance. Certains utilisent des approches adaptatives afin de détecter les problèmes de manière proactive. D'autres se concentrent davantage sur l'analyse des causes profondes (root cause analysis). Ces approches peuvent être classées selon les problématiques de latence des requêtes, les méthodes de reconnaissance de motifs, etc.

CauseInfer [39] et Sieve [40] s'appuient sur la construction de graphes causaux représentant les différents composants du système, utilisés ensuite pour identifier les causes sous-jacentes. Cependant, ces techniques ne permettent pas de déterminer précisément la direction de la dépendance entre deux instances de services.

MicroScope [41] utilise l'algorithme PC pour créer un graphe des dépendances entre services, mais se limite aux traces anormales, négligeant les informations pouvant être extraites des traces normales.

MicroRank [42] propose une méthode basée sur l'analyse spectrale et l'algorithme PageRank. Elle distingue différents types de requêtes et attribue un poids à chacune en fonction de sa capacité à localiser les causes racines. Elle accorde une attention particulière aux requêtes peu fréquentes pour assurer une meilleure couverture, mais ne répond toujours pas à la question de savoir pourquoi certaines requêtes sont plus lentes.

ADSketch [43] est une méthode de détection d'anomalies basée sur la reconnaissance de motifs. Elle est interprétable et adaptative, identifiant des groupes de patrons de métriques anormaux représentant différents types de problèmes. Cependant, elle ne distingue pas les types de requêtes ni n'explique pourquoi les anomalies surviennent.

Huang et al. [44] ont proposé un arbre de variance pour localiser les problèmes dans les systèmes mono-nœud. Cela permet de comprendre comment la variance globale de latence est liée aux variances et covariances des temps d'exécution de fonctions spécifiques. Ils ont développé VProfiler, un outil de profiling permettant d'identifier les principaux contributeurs à la variance du temps d'exécution d'une tâche, à partir de l'analyse du code source. Cet

outil est toutefois limité aux applications C/C++.

2.8 Approches modernes de l'intelligence artificielle

Certains chercheurs [45,46] ont utilisé des techniques d'analyse de traces basées sur l'apprentissage profond pour identifier des anomalies dans les systèmes à microservices. Ces méthodes considèrent une trace comme une séquence d'appels de services, mais ne tiennent pas compte des configurations complexes induites par la hiérarchie des appels ou les invocations parallèles/asynchrones.

D'autres approches [47–49] s'appuient sur les journaux (logs) pour comprendre le comportement des services. Elles apprennent les motifs de logs lors d'une exécution normale et détectent les anomalies par déviation par rapport au modèle.

Liu et al. [50] ont proposé une méthode d'extraction de caractéristiques spatio-temporelles basée sur la dynamique symbolique, combinée à un modèle de Boltzmann restreint (RBM) pour détecter les anomalies, et analyser leur propagation afin d'identifier les causes profondes. Par la suite, [51] ont introduit un modèle de transition d'états séquentiels avec RBM et une association artificielle d'anomalies.

Li et al. [52] ont utilisé un modèle latent dynamique et une analyse de causalité basée sur le Dynamic Time Warping pour identifier les causes racines. Cortellessa et al. [53] ont introduit une méthode d'apprentissage automatique pour détecter la dégradation des performances, en regroupant les traces d'exécution similaires. Les exécutions qui s'écartent des regroupements (clusters) sont alors identifiées comme anormales. Leur méthode utilise un algorithme génétique guidé par une métrique de pertinence et une fonction d'évaluation optimisée, permettant d'identifier les groupements présentant des motifs similaires de dégradation de la latence dans les appels RPC.

Dymshits et al. [54] ont proposé une méthode de surveillance basée sur l'apprentissage de séquences de vecteurs de comptage d'appels système, à l'aide d'un réseau LSTM (Long Short-Term Memory). Le modèle détecte les changements de comportement de processus, pouvant résulter d'une activité malveillante ou d'autres dysfonctionnements. Le LSTM, étant un type de réseau de neurones récurrent, est adapté à l'analyse de séquences de longueurs variables.

Fournier et al. [55] ont proposé une méthode pour détecter les anomalies de performance dans les requêtes web, en identifiant leurs causes profondes via le comportement interne. Ils collectent des traces en espace utilisateur et noyau à l'aide de LTTng, et extraient depuis le chemin critique de chaque requête des séquences d'appels système, des séquences d'états d'exécution, le nombre d'occurrences et la durée totale des états. Ces caractéristiques sont en-

suite analysées par un détecteur d'anomalies basé sur le clustering DBSCAN. Les techniques BOW et TFIDF sont utilisées pour distinguer les événements rares, et les valeurs extrêmes sont regroupés par comportement pour une analyse détaillée.

Toutes ces approches nécessitent une compréhension approfondie du système et une maîtrise des nombreux paramètres impliqués dans les techniques d'apprentissage utilisées.

Les méthodes traditionnelles d'analyse des journaux (logs) et des traces rencontrent souvent des difficultés face aux volumes de données massifs et aux patrons complexes. Elles ne sont pas adaptées à la croissance du volume, de la diversité et des besoins en raisonnement profond dans les systèmes modernes. Les modèles de langage de grande taille (LLMs) offrent une alternative puissante en surmontant bon nombre de ces limites, notamment grâce à leur capacité à comprendre le contexte et à détecter les relations causales. Toutefois, malgré leurs avantages, les LLMs présentent également leurs propres limites.

Les LLMs ont récemment apporté des améliorations majeures aux tâches de traitement du langage naturel (NLP) [56, 57]. Ils génèrent du texte en prédisant chaque mot à partir des précédents, selon une méthode statistique et autorégressive. Cela les rend très efficaces pour produire des réponses pertinentes et cohérentes [58]. Néanmoins, leur application à des problèmes spécifiques requiert souvent des techniques d'adaptation supplémentaires.

Pour personnaliser les LLMs selon les tâches, les chercheurs utilisent des méthodes comme le raffinement (fine-tuning) ou l'apprentissage en contexte (*in-context learning*, ICL). L'ICL est une solution populaire et économique. Elle consiste à fournir au modèle quelques exemples en entrée, généralement en langage naturel. Ces exemples aident le modèle à comprendre la tâche à accomplir. Le modèle utilise ensuite cette entrée, combinée à la tâche, pour générer un résultat [59]. Ce processus évite le recours à un ré-entrainement coûteux et permet une utilisation à grande échelle [59, 60].

L'apprentissage à quelques exemples (*few-shot learning*) repose sur l'ICL en permettant au modèle d'apprendre à partir d'un nombre très limité d'exemples. Le *Chain of Thought* (CoT) améliore le raisonnement du modèle en décomposant les réponses en étapes logiques. Le *Zero-shot-CoT* pousse encore plus loin cette capacité, en permettant au modèle de résoudre des tâches complexes sans aucun exemple, simplement via des entrées comme "Réfléchissons étape par étape" [61]. LogRules [62] est un cadre léger basé sur des règles pour l'analyse des journaux, qui améliore les performances des petits modèles de langage tout en visant un faible coût d'inférence et une meilleure généralisation. Toutefois, il présente des limitations liées à l'utilisation des jetons et à l'efficacité des coûts : l'intégration de multiples règles dans les invites peut entraîner un dépassement du nombre de jetons, notamment avec des journaux longs ou un grand nombre de règles, dépassant ainsi les limites de contexte des

modèles plus petits. De plus, afin de maintenir un coût d'inférence abordable, le cadre doit limiter la longueur des invites, ce qui peut réduire la complexité ou le nombre de règles utilisables, affectant potentiellement l'efficacité du raisonnement. Les LLMs peuvent être classés en trois types d'architectures : encodeur seul, décodeur seul, ou encodeur-décodeur. Chaque architecture est adaptée à des types de tâches spécifiques. Les modèles à décodeur seul, comme GPT, excellent dans les tâches *few-shot* et *zero-shot* grâce à leur conception autorégressive. Cela les rend particulièrement efficaces pour la génération de texte et la réponse aux questions [63, 64]. Dans notre travail, nous avons choisi GPT-4o car il gère efficacement l'analyse de logs complexes et contextuels.

Bien que les LLMs résolvent de nombreux problèmes des approches traditionnelles, ils présentent également certaines limitations. L'une des principales est la taille de la fenêtre de contexte — la quantité maximale de texte que le modèle peut traiter en une fois [57, 63, 65]. Lorsque cette limite est dépassée, le modèle peut perdre en précision ou se comporter de manière imprévisible. Il est donc crucial de bien gérer la taille des entrées, pour des données complexes et volumineuses.

En résumé, les LLMs offrent des avantages significatifs pour l'analyse et la compréhension de données complexes issues des journaux systèmes, notamment grâce à une meilleure gestion du contexte et du raisonnement causal. Néanmoins, ils posent encore des défis, tels que les limites de fenêtres de contexte, les besoins matériels élevés, ainsi que le coût et la latence liés à l'utilisation d'APIs externes. Pour assurer un usage fiable et évolutif, ces facteurs doivent être rigoureusement gérés et optimisés.

2.8.1 Approches basées sur les graphes

Les méthodes de cette catégorie consistent généralement à construire un graphe de dépendances entre les nœuds d'un système distribué, afin d'établir des relations entre composants avant d'effectuer le diagnostic ou l'analyse. Ces techniques exploitent également des métriques de surveillance pour détecter les anomalies et construire des graphes de causalité selon différentes approches. Le diagnostic des problèmes s'appuie ensuite sur l'analyse combinée des métriques et du graphe de dépendances [42].

CauseInfer [39] crée un graphe hiérarchique de causalité à deux niveaux représentant le système distribué, puis utilise des méthodes statistiques pour inférer les causes racines des problèmes de performance le long du graphe. Sieve [40] applique le test de causalité de Granger pour déduire les dépendances entre les métriques de performance des composants distribués. Dans Microscope [41], les appels système et les sockets réseau sont interceptés pour obtenir les dépendances réseau, qui sont ensuite utilisées pour construire un graphe de

dépendances. Pour réaliser une Root Cause Analysis (RCA), Microscope compare la similarité entre les métriques SLO (Service Level Objective) et les services anormaux afin d'identifier les candidats aux causes racines.

L'avantage principal de ces méthodes est qu'elles ne nécessitent pas d'instrumentation du code source. Cependant, elles fournissent en général moins de détails pour l'analyse que les méthodes basées sur le traçage.

2.9 Conclusion

De manière générale, plusieurs défis doivent être relevés pour évaluer efficacement la dégradation des performances, tous issus de problèmes concrets et des retours d'expérience de notre partenaire industriel :

- **Comportement dynamique des logiciels :** les systèmes logiciels présentent des comportements dynamiques et variables, comme une utilisation fluctuante de la mémoire, en fonction du comportement des utilisateurs et de la charge du système. Il est donc difficile de décrire avec précision le comportement d'un système logiciel et de capturer fidèlement son état pré-défaillance [66]. Les méthodes existantes reposant sur des modèles statistiques peuvent classer à tort une utilisation fréquente des ressources comme un comportement anormal [67–70]. En outre, elles ne permettent pas de distinguer les différences entre plusieurs exécutions.
- **Défis d'instrumentation :** il n'est pas toujours possible pour les développeurs d'accéder au code source dans les systèmes infonuagiques ou ceux utilisant des modules tiers, ce qui rend l'utilisation des techniques classiques de détection des anomalies de performance peu pratique [71, 72]. L'ajout de points d'instrumentation dans le code source ou binaire peut entraîner une surcharge importante et affecter les performances du système à l'exécution [72, 73].
- **Défis liés à l'acquisition des données :** il n'est pas toujours possible de disposer de données étiquetées en quantité suffisante pour utiliser des approches d'apprentissage automatique [74–76]. La plupart des entreprises appliquent des politiques strictes en matière de sécurité et de confidentialité, ce qui limite la mise à disposition de leurs données. Par conséquent, ces techniques sont souvent limitées à la détection d'anomalies connues, observées durant les phases d'entraînement [77].

CHAPITRE 3 APERÇU GÉNÉRAL

Cette thèse est structurée autour de trois articles, présentés dans l'ordre de leur soumission pour publication, dans les Chapitres 4, 5 et 6. Chaque article répond à un objectif de recherche différent introduit au Chapitre 1, et ensemble ils forment une approche cohérente pour résoudre un problème commun.

L'objectif principal de cette thèse est de faciliter le diagnostic des dégradations de performance dans les systèmes distribués. Elle vise à automatiser, rendre plus efficace et plus scalable les processus de comparaison et d'analyse des causes profondes. Pour ce faire, ce travail combine des techniques de traçage, de mesure de performance système, et de traitement des journaux afin d'améliorer l'observabilité des systèmes et la précision des diagnostics dans des environnements distribués complexes.

Ce qui suit est un aperçu succinct de chaque article :

Chapitre 4

Ce chapitre présente l'article intitulé « DTraComp : Comparing Distributed Execution Traces for Understanding Intermittent Latency Sources », soumis au Journal of Systems and Software. Cet article introduit `DTraComp`, un nouveau cadre en logiciel libre conçu pour permettre aux ingénieurs de comparer les données de traçage distribuées et de détecter les dégradations de performance dans les systèmes à microservices. `DTraComp` intègre un traçage distribué de haut niveau avec des événements noyau de bas niveau et visualise les différences à l'aide de graphiques en flammes différentiels enrichis. Il fournit des métriques précises telles que les temps d'attente CPU et E/S pour chaque intervalle (span), et prend en charge l'analyse des exécutions concurrentes. En facilitant la comparaison côte à côte des traces normales et anormales, cette méthode simplifie l'identification des sources de latence intermittentes et des goulets d'étranglement. Intégré à Eclipse Trace Compass, l'outil établit une base pour une analyse des traces distribuées systématique et scalable.

Chapitre 5

Ce chapitre présente l'article intitulé « Lightweight Root Cause Analysis for Latency Issues Using Critical Path-Aware Hybrid Tracing ». Ce travail s'appuie sur les fondements posés au Chapitre 4 en introduisant une méthodologie plus ciblée et efficace pour l'analyse des causes profondes. L'approche proposée identifie automatiquement le chemin critique des requêtes

distribuées—la plus longue chaîne de dépendances qui dicte la latence de bout en bout—et concentre l'analyse uniquement sur les intervalles (spans) ayant un impact direct sur les performances. Elle combine le traçage des intervalles (spans) de haut niveau avec une collecte sélective de métriques système de bas niveau, afin de minimiser la surcharge en ressources. En outre, elle utilise la synchronisation des horloges, la localisation de fautes basée sur le spectre et le classement par Personalized PageRank pour classer et isoler avec précision les services problématiques. Il en résulte un cadre adaptatif et automatisé qui améliore la précision des diagnostics tout en réduisant significativement le temps d'analyse et la consommation de ressources système.

Chapitre 6

Ce chapitre présente l'article intitulé "InsightAI : Root Cause Analysis in Large Log Files with Private Data Using Large Language Model." Bien que cet article se concentre sur l'analyse des causes profondes à partir des journaux d'événements, l'approche n'est pas limitée à ces derniers ; elle peut également être appliquée aux traces et aux métriques système. Cependant, les besoins de notre partenaire industriel portaient spécifiquement sur les journaux. Il introduit `InsightAI`, un système de chatbot respectueux de la vie privée, qui exploite des modèles de langage de grande taille (LLM) pour analyser de vastes ensembles de journaux via des requêtes en langage naturel. Le système met en œuvre une stratégie de compression inspirée des graphiques en flammes pour réduire le nombre de jetons de plus de 90 %, améliorant ainsi l'efficacité et la rentabilité. Il applique également des techniques d'anonymisation pour protéger les données sensibles et utilise la classification des intentions ainsi que des méthodes de recherche intelligente pour extraire uniquement les segments de journaux les plus pertinents. Déjà utilisé dans l'industrie, `InsightAI` constitue une extension concrète de cette thèse en démontrant comment les techniques basées sur l'IA peuvent renforcer l'observabilité et automatiser l'analyse des causes profondes, à partir des journaux, dans les systèmes distribués.

Ensemble, ces trois contributions forment un cadre complet permettant aux ingénieurs de détecter, comparer et comprendre automatiquement les dégradations de performance dans les systèmes distribués, en couvrant la visualisation des traces, la détection du chemin critique, et l'interprétation intelligente des journaux.

CHAPITRE 4 ARTICLE 1: DTRACOMP COMPARING DISTRIBUTED EXECUTION TRACES FOR UNDERSTANDING INTERMITTENT LATENCY SOURCE

Authors Maryam Ekhlasi, Fatemeh Faraji Daneshgar, Maxime Lamothe, Michel Dagenais, Naser Ezzati-jivan, Matthew Khouzam

Accpeted with major modifications at the Journal of Systems Software (25 July 2025)

Abstract Microservice architectures significantly enhance software development by offering flexibility in programming languages and deployment infrastructures. They facilitate the isolation of failures within individual services and simplify the process of debugging and fixing issues in independent services. Nonetheless, pinpointing performance degradation becomes a challenging task due to the complex interactions and parallelism among numerous service instances. While end-to-end tracing aids in mapping execution paths across services and identifying latencies, its ability is limited to gathering high-level information, not fully capable of pinpointing the root causes of performance degradation among processes. Additionally, a notable gap in many current performance analysis tools is the absence of a comparison feature, which is crucial for providing developers with a detailed perspective on the performance variations between different sets of requests. This paper introduces *DTraComp (Distributed Trace Compare)*, an open-source framework, compatible with various microservice trace standards, and integrated with Eclipse Trace Compass™ [1]. The demo is available on YouTube [2]. Our framework offers robust visual comparison capability for two groups of requests within distributed systems, which includes nested spans executed in parallel. Furthermore, it provides system kernel details for each thread involved in the execution of each span, allowing it to pinpoint the reasons for performance degradation across distributed systems. We used our proposed framework to analyze five practical use cases. By evaluating the efficiency of our tool, it was determined that the overall time complexity scales linearly O(n) with the trace size, indicating its suitability for deployment in production environments. It is currently used within the Ericsson company for performance evaluation purposes.

4.1 Introduction

Modern businesses benefit from implementing microservice architectures, with improved scalability of components, increased productivity among developers, and reduced limitations re-

1. `https://projects.eclipse.org/projects/tools.tracecompass`
2. `https://www.youtube.com/watch?v=X3CfAlWllDc`

lated to programming languages [41, 78–80]. In a microservice system, every service operates as a group of instances, on one or more machines, and interacts with other microservices through lightweight mechanisms [78]. However, the complex and dynamic nature of a microservice environment leads to several issues, such as hardware failures, improper configurations, implementation flaws, and incorrect coordination between services, causing them to fail or experience performance degradation [81, 82] leading to a more complex root cause localization.

Distributed tracing simplifies the tracking of user requests, in distributed applications, by propagating a tracing context across nested sub-requests. Distributed tracing techniques, while effective for high-level analysis, lack the granularity to provide insight into system-level metrics such as CPU usage or disk contention. Conversely, software tracing captures kernel- and user-level events, providing detailed system-level information for diagnosing bugs and latency. It excels at high-throughput event collection, with a low impact on the target application. However, software tracing is not designed for end-to-end tracing in distributed environments [83].

Building on this limitation, another layer of complexity exists: in a complex system, low-level events such as CPU usage or disk reads/writes occur at the kernel or system level. These are crucial for performance but often do not directly map to high-level operations, such as a user request to view a webpage. The challenge is to link these two levels of operation in a meaningful manner. Without this link, users may know that the CPU spiked or that there was heavy disk usage, but not what that means in terms of user experience or business logic.

Distributed systems faces another problem. Distributed systems often produce large and complex sets of trace data. Parsing through these data to determine the cause of a performance issue is not an easy task. This challenge is escalated by the fact that the architecture is usually complex and involves multiple services and components that interact in intricate ways. It is not just about having a lot of data; it is about having complex, interconnected data. This is time-consuming for users, as they have to go from high-level end-to-end tracing data down to low-level system information to find the root causes, mostly by manually switching between tools or views. Besides, diagnosing an issue requires a deep knowledge of how services communicate and are structured.

Given the challenges of both distributed tracing and software tracing, especially when managing tracing at both the userspace and kernel levels for each software application, there is a critical need for a comprehensive solution. To address these challenges, we introduce a visualization framework that integrates with Eclipse Trace Compass™ [3], a popular open-source

3. `https://projects.eclipse.org/projects/tools.tracecompass`

tool for trace analysis and visualization. This framework combines distributed tracing with kernel-level events, for each machine, synchronizing them to provide a unified and detailed view of system performance.

The goal is to present a visualization tool that enables users to compare different requests across various levels, from high-level overviews to detailed low-level analyses, regardless of whether the traces come from a single machine or multiple machines. It is compatible with widely used microservice trace standards, as shown in Figure 4.1. The tool automatically highlights differences between two sets of similar requests and calculates various performance metrics for each service. This enables users to quickly detect and diagnose the root causes of performance degradation, from high-level system overviews down to low-level details in both userspace and kernel-level events. This method enhances the observability of distributed systems and makes it easier to diagnose performance degradation in complex, large-scale traces.

We enhanced our framework by adding a differential flame graph, making it easier to spot differences between similar requests. This addition effectively highlights variations in performance degradation across different spans in distributed traces. A span [84] in distributed tracing is a fundamental unit of work or operation within a trace, representing a specific segment of processing, such as a single function execution, thread execution, or remote procedure call (RPC).

Our framework enables visual comparison between two groups of requests in distributed systems, even when nested spans are executed in parallel. It extends beyond basic visualization by providing detailed kernel-level information for each thread on each machine. This level of detail allows for precise identification of the root causes of performance degradation. Compared to existing solutions, our framework provides a deeper level of detail that not only enables in-depth analysis but also facilitates troubleshooting and optimization.

The contributions of this study are as follows.

— DTraComp is a user-friendly, open-source tool that helps experts identify and perform root cause analysis of performance problems in various types of applications, whether they are distributed or standalone systems. It is capable of analyzing both sequential and concurrent or parallel services, making it suitable for a wide range of use cases. The source code is available on GitHub [4], and a demo can be viewed on YouTube [5].

— Our framework offers several independent views. Each module, as explained in Section

4. `https://git.eclipse.org/r/c/tracecompass.incubator/org.eclipse.tracecompass.incubator/+/196496`

5. `https://www.youtube.com/watch?v=X3CfAlWllDc`

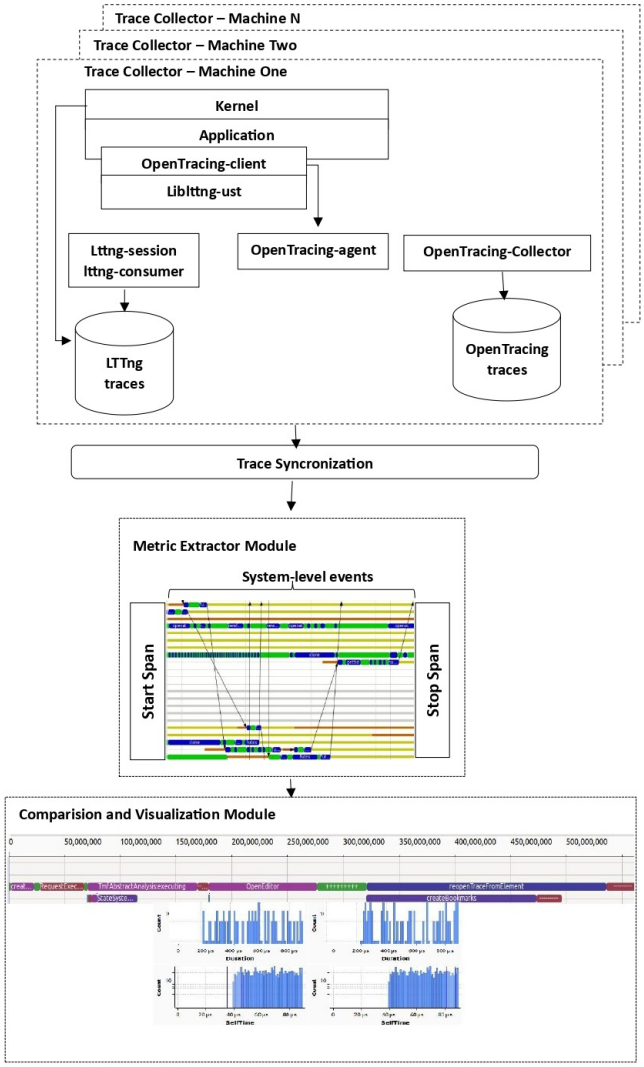


Figure 4.1 DTraComp Software Architecture. The architecture is explained in detail in Section 4.3.

4.3, can operate on its own but also complements the others by providing a layered view of information, ranging from a high-level overview to detailed low-level data.

— It enables visual comparison between two groups of requests in distributed systems, even when nested spans are executed in parallel.

— It extends beyond basic visualization by providing detailed kernel-level information for each thread involved in running spans or services on each machine such as CPU wait time, I/O wait time, etc. This level of detail allows for precise identification of the root causes of performance degradation.

  — Each span can be executed by several threads, and each thread has several states during its lifetime. We have computed the precise duration of each system state

for all threads participating in each span, enabling experts to perform root cause analysis and identify the reasons for performance degradation.
  - To identify the exact cause of latency in a specific system state, we measured the latency of each system call associated with each span, enabling us to determine the primary factor responsible for performance changes.
- Our framework is integrated into the Eclipse Trace Compass application, supporting various tracing formats and enabling the use of additional features provided by this powerful tool.
- Our Trace Collector module, described in Section 4.3.1, introduces minimal overhead, even when handling high-volume traces. Additionally, the execution time of our modules that calculate metrics and visualize charts remains negligible, ensuring that no additional overhead is imposed on the system.
- We illustrated how DTraComp can diagnose the performance problems of five real open-source popular distributed applications in Section 4.4.
- We have introduced a hybrid approach of trace collecting in Section 4.3.1 capable of extracting low-level kernel events corresponding to each high-level span in the distributed systems

The remainder of this paper is organized as follows. In Section 2, we present a comprehensive review of prior research focused on detecting and diagnosing performance problems. In this section, we clarify how our approach can effectively address the limitations observed in previous studies. Moving on to Section 3, we provide an in-depth discussion of our proposed solution, while Section 4 offers a demonstration of the ability of our framework to diagnose five real-world performance problems. In Section 5, we evaluate each module within our framework and compare it with other similar approaches.

4.2 Related Work

Over the past few years, extensive research has been conducted on detecting and diagnosing performance degradation, enabling categorization based on low-level tracing, end-to-end tracing, and hybrid methodologies. Meanwhile, some researchers have focused on comparing two groups of requests or executions to facilitate the process of problem diagnosis. Below, we discuss some pertinent research in the context of our proposed framework.

4.2.1 High-level End-to-end Tracing Approaches

MicroRank [85] proposed a method based on spectrum analysis and a page rank algorithm for root-cause localization. It can discriminate between different requests and weigh each request based on its ability to localize the root causes. It also pays closer attention to the less frequently occurring types of requests, to achieve a more balanced distribution of request types. However, their method is unable to explain why some requests take longer than their peers, and do not provide a module for comparison purposes.

VProfiler [44] is a profiling tool that can accurately identify the main factors contributing to the variance of the execution time of a particular function, in single-node systems, by analyzing the application source code. This tool can provide a comprehensive understanding of the system performance. However, it is limited to C/C++ monolithic applications.

MicroScope [41] uses a PC algorithm to create a graph that reflects the dependencies between services. It compares the similarity between service-level objective metrics and abnormal services to find the root cause candidates. The advantage of these methods is that they do not require the source code to be instrumented. However, they may not be able to offer detailed data for analysis, compared to trace-based methods, because they only focus on abnormal traces, neglecting the insights obtained from normal traces.

ADSketch [43] is a performance anomaly detection method that employs pattern sketching. It is both interpretable and adaptable. It identifies groups of anomalous metric patterns, each representing a particular type of performance problem. However, it is not possible to distinguish between the various types of requests, and cannot explain why performance issues occurred.

Some researchers [45, 46] used trace analysis techniques, based on deep learning, to identify anomalies during the runtime of microservice systems. These methods consider traces to be a sequence of service calls. Nevertheless, traces can have intricate configurations that arise from the hierarchy of service calls and parallel or asynchronous invocations. These methods do not consider the complex relationships between traces.

Du et al. [49] proposed DeepLog, an LSTM-based model that learns normal log patterns and detects anomalies when logs deviate from these patterns. It supports online updates by incorporating user feedback, requiring periodic retraining of the model.

Meng et al. [48] proposed LogAnomaly, a framework that models log streams as natural language sequences. Using template2vec, a method for extracting hidden semantic information in log templates, LogAnomaly detects both sequential and quantitative log anomalies. However, in online detection, real-time logs that deviate from learned patterns are flagged as

anomalies.

Zhang et al. [47] introduced LogRobust, which extracts semantic information from log events and represents them as vectors. It uses an attention-based Bi-LSTM model to detect anomalies by capturing contextual information and learning the importance of log events. However, sudden or large-scale changes in logs or logging mechanisms remain challenging.

Liuet et al. [50] suggested a method for identifying causal relationships using symbolic-dynamics-based spatial-temporal feature extraction. To detect anomalies, they used a Restricted Boltzmann Machine and analyzed the propagation of these anomalies to determine the underlying causes.

Li et al. [52] used a dynamic latent variable model, and a causality analysis technique based on dynamic time warping, to identify the root causes. Cortellessa et al. [53] introduced a method that used machine learning to detect performance degradation by clustering similar execution traces. Unlike our proposed approach, all the aforementioned techniques require a deep understanding of the system, and a variety of parameters involved in the training techniques.

Beschastnikh et al. [86] introduced ShiViz, a tool that visualizes distributed system executions through time-space diagrams, along with XVector, a library that incorporates vector timestamps into system logs to capture event order. These tools have proven effective in helping developers understand, query, and compare the behavior of distributed systems, as highlighted in various studies. While ShiViz is a powerful tool, it currently lacks a comparison feature for analyzing different executions side by side, which would be valuable for root cause analysis. Additionally, it does not provide detailed information on performance degradation or the underlying issues causing such problems.

TraVista [87] is a tool developed for debugging performance issues within a single trace. It enhances the widely used single-trace Gantt chart visualization by integrating three types of aggregated data—metric, temporal, and structural to provide context for the performance of the problematic trace in relation to all traces. While TraVista excels in diagnosing performance issues within individual traces, it currently does not support the comparison of two sets of executions, which could enhance its ability to fully diagnose issues arising from differences between multiple trace sets.

VAMP [88] is a visualization tool for performance analysis that depicts end-to-end tracing using a Tree diagram and Histogram charts. It aggregates RPC nodes and highlights execution time differences with color coding. The tool includes a histogram that allows users to filter requests based on execution time. However, users cannot select and compare specific

requests directly; they can only filter based on latency. While VAMP highlights spans with potential issues, it does not support self-time (exclusive execution time) analysis and cannot provide detailed system information for root cause analysis.

4.2.2 Low-Level Software Tracing

The techniques mentioned above are specifically suitable for distributed systems because they emphasize context propagation and large-scale tracing. However, it is essential to note that these techniques are limited to capturing high-level events [83] and cannot extract system information related to performance degradation. Researchers [89, 90], who work on low-level traces to detect or diagnose software performance degradation, encounter a large volume of events that they need to collect with high accuracy and low overhead.

Dymshits et al. [54] introduced a method for monitoring processes that relies on a learning model for system call sequences. They continuously recorded the sequence of system calls as they occurred in real time by Sysdig [91]. Their analysis aimed to identify abnormal activities within a process. To achieve this, they employed a multiclass classification technique, tailored to the stream of system calls. This approach can process a large volume of kernel events, but cannot provide performance information on requests in distributed systems. To evaluate the performance of distributed systems, we need a tool that can collect traces with low overhead, at both the user and kernel levels.

LTTng is an open-source, low-overhead, high-performance tracing tool that allows the integrated analysis of kernel space and user space events simultaneously. Because it uses approximately 2% of the CPU time on a heavy workload [92], it is an ideal tool for low-level trace collection.

Most low-level performance analysis research is challenged by the high volume of system events. To address this problem, Montplaisir-gonçalves et al. [93] proposed a state history tree, where a special index of state intervals is constructed progressively, as a trace is being analyzed. This structure records details such as system performance metrics since the trace began. The data is organized to facilitate the quick retrieval of attribute values at specific times, even when the state history exceeds the capacity of the main memory. Eclipse Trace Compass™ is a trace analysis tool that uses this data structure for interactive views. State history trees enable the evaluation of system metrics for any desired time interval, with logarithmic time complexity [33, 94]. Using this state history, Eclipse Trace Compass is a versatile and efficient tool for trace analysis.

4.2.3 Hybrid Software Tracing

Since low-level and high-level software tracing offer unique advantages, hybrid approaches have been proposed to make the best of both worlds. Gelle et al. [83] suggest a solution that combines kernel tracing with distributed tracing, to better scrutinize events and determine the underlying cause of performance issues. By merging the advantages of both approaches, this method can obtain precise information about system interactions and high-level events, from distributed tracing, while collecting detailed and specific low-level events from kernel and user space tracing. However, in contrast to our approach, this proposed approach does not provide an automated comparison module. Furthermore, it focuses only on the critical path for each execution.

Kohyarnejadfard et al. [95] proposed a framework for detecting and localizing the performance anomalies. It extracts the sequence of the kernel events for each span and then detects anomalous behavior based on NLP techniques. Unlike our approach, this approach does not contain a module to compare the differences between the groups of executions. Furthermore, it requires deep knowledge of the application for grouping kernel events related to each application span.

Fournier et al. [55] proposed a method for detecting performance anomalies in web requests, and identifying their root causes, by examining their internal behavior. This technique involves collecting user space and kernel space traces of web requests. Unlike our approach, their proposed approach focuses only on the start and end points of each request, and cannot differentiate the spans within each request.

4.2.4 Comparing Task Executions

Visual solutions allow us to compare requests in terms of implementation, hardware, or system configuration changes, and thus quickly identify performance problems. After localizing the performance degradation, we proceed to the root cause by extracting more features to diagnose the exact problem. Visualization can help to compare the performance profiles of two different executions within a software system.

The differential flame graph [96] effectively shows the differences between two CPU profiles. It visually represents call stacks in the y-axis, whereas the size and shape of the flame graph show the second profile (current or after). In addition, the color of each rectangle indicates the contrast between the time spent in the associated call stack in the two profiles. Colors such as red and blue indicate that the load increases and decreases, respectively. Although this view helps compare the performance of two different profiles, it is limited to the analysis

of CPU consumption.

Tracecompare [33] introduced a new data structure called an enhanced calling context tree (ECCT) to represent all latencies that affect the duration of task execution. This data structure enables it to determine the latency of executions containing off-CPU wait times and multithreaded executions. An enhanced differential flame graph was introduced to visualize the comparison. The comparison was limited to identical executions, and the approach struggles when the program encounters significant code changes. Furthermore, this approach cannot extract valuable metrics from user space processes, especially with end-to-end tracing in distributed systems.

Raja et-al. [97] introduced a side-by-side visualization tool that compares two requests (before and after) by displaying them as graphs, allowing users to observe latency changes. However, this tool lacks the ability to explain why performance degradation occurs, and cannot compare two sets of similar requests.

According to the research conducted by Davidson et al. [84], which explored user requirements for visualization tools in distributed tracing, we have developed a distributed visualization tool that addresses several of the key challenges users face with such tools. Specifically, it provides advanced trace comparison features for efficiently comparing individual or groups of requests through distributed systems, highlighting spans based on their latency, and provides a hierarchical view that enables users to look at the traces from a high level and dig into low level performance metrics for each span or services.

4.3 Proposed Solution

One of the main challenges for developers and engineers is understanding how the software performance changes after modifications like versioning, hardware updates, or configuration changes. This becomes even more challenging in distributed systems, where many machines work independently, and some performance degradation can propagate to other related services, affecting the entire system performance. Even in standalone applications, some requests can take longer to execute than expected. To address these challenges, comparison tools are essential. They allow users to compare different executions and identify differences. By comparing, experts can find out which services have longer response times than usual. We have developed a framework to find the suspicious service or services, then dig into more detailed information related to the execution of the selected service.

Our visualization framework, as shown in Figure 4.2, provides a top-down approach addressing the mentioned challenges. At the highest level, our filtering module (1) presents

two groups of similar executions based on their duration and self-time. Self-time refers to the amount of time a service spends executing its own code, excluding any time spent in child services, functions, or external calls that it may invoke. Duration, on the other hand, is the total time taken by the service to complete an operation, including the time spent in self-processing as well as the time spent waiting on or interacting with other services or functions.

This module allows users to adjust the range for comparison by dragging the desired range. Below this (2), we list all similar requests with checkboxes, enabling users to manually select specific requests if needed. Then, a flame graph is generated for each selected request. Flame graphs within each group are merged forming a general flame graph. So at the end of this step we have two merged flame graphs, FG_A and FG_B. To form the Differential flame graph in part (3), FG_A is considered as the based flame graph in which the captions are updated. The updated captions represent the differential ratio between the corresponding spans between FG_A and FG_B. The details of differential ratio calculation is explained in Subsection 4.3.3.

The differential flame graph is a powerful tool where each flame or box represents a service or span, with its width corresponding to the ratio of the duration time. By merging flame graphs from multiple spans within a group, we create a comprehensive visual representation. This unique representation not only highlights disparities between the two groups but also emphasizes the most critical differences using distinct color tints, making it easier to pinpoint significant performance variations. The enhanced differential flame graph further clarifies these key differences, making comparisons more insightful and easier to interpret. Our Differential Flame Graph allows users to observe overall performance characteristics and identify patterns or common bottlenecks within the group.

Once a problematic span is detected, our Metric Extractor module in part (4) helps experts in diagnosing the performance issue and conducting root cause analysis. In this section, we list all spans along with both high-level and system-level information, including Start Time, End Time, Span ID, Trace ID, System Process ID, System Thread ID, and the execution time of the system states related to each span. This includes detailed metrics such as CPU wait time, memory wait time, I/O wait time, and more granular data like the time taken for read, write, network operations, and other system activities. This detailed information supports a thorough analysis, enabling the root cause analysis of performance problems. The source code for this module is available on GitHub [6]. And the application is now live in Eclipse

6. `https://git.eclipse.org/r/c/tracecompass.incubator/org.eclipse.tracecompass.incubator/+/203508`

Trace Compass [7]. In the following subsections, we provide a detailed explanation of each module.

4.3.1 Trace Collector

To achieve a comprehensive and precise top-down comparison of the performance of distributed requests, we integrated high-level end-to-end traces with low-level kernel events. This is explained in more details in the following subsection.

Hybrid Tracing

We used a double instrumentation approach to combine high and low-level tracing. Although our framework can easily adapt to any framework designed for tracing distributed systems, we selected OpenTracing instrumentation, to evaluate the overhead of our trace collector and compare it to previous research [83].

We used the HotROD application, already instrumented with OpenTracing (the first instrumentation), to capture high-level spans. Then, we instrumented the OpenTracing source code with LTTng [8] (the second instrumentation) to capture the start and end points of each span. LTTng allows us to collect user-space events (UST) along with kernel events, with a unified clock, and brings low instrumentation overhead.

LTTng allows simultaneous tracing of both userspace and kernel events by creating separate channels: one for userspace events (userchannel) and another for kernel events (kernelchannel), meaning we can track the start and end of each span and all the kernel events that occur in between. It captures all relevant events within these channels, adding context such as process and thread IDs, to ensure that each event is linked to its corresponding process or thread. LTTng-UST (User Space Tracer) integrates with the root tracer and root logger of a logging framework within an application. This integration allows any tracepoint, including those from the OpenTracing library, to generate LTTng events. By adding a handler to the root logger, LTTng-UST captures all traces produced by the application.

This instrumentation capability enables us to inject critical metadata—such as Start Time, End Time, Span ID, Parent ID, and Distributed Trace ID—into each tracepoint of the OpenTracing library. At the same time, by capturing the system thread ID and process ID for each tracepoint, we establish a correlation between high-level and low-level traces. The collected events are then sent to the LTTng backend for recording and further analysis.

7. `https://www.youtube.com/watch?v=X3CfAlWllDc`
8. Linux Trace Toolkit Next Generation `https://lttng.org/`

Since each span can be executed by multiple system threads. We capture the start and end events of each span using UST events, which include the thread ID, process ID, start time, and end time. Then from these events, we track all kernel events that occur during the execution of a span. As the threads transition through states like running, waiting, blocked, or sleeping, we measure the time spent in each state, using kernel events to calculate the duration of each. This detailed performance data is crucial for identifying bottlenecks, such as threads waiting for I/O operations, CPU utilization, or other resource constraints.

Because each machine in a distributed system operates on its own clock, we need to synchronize and integrate traces and it involves multiple synchronization steps. These steps include matching traces from the higher and lower levels within each machine and then integrate them all together. We added our framework to Eclipse Trace Compass™ to use its ability to synchronize traces from different machines. In the next section, we provide a more detailed explanation.

4.3.2 Trace synchronization

System traces from distributed applications are collected from different machines, and since each machine operates with its own clock, scaling the tracing approach involves recording traces on each node and then merging and analyzing them. To make this process effective, it is crucial to ensure that event timestamps are accurately synchronized across all machines in the distributed system. For instance, tracing can capture events with extremely precise nanosecond accuracy, while network latency typically achieves an accuracy within microseconds [98]. This synchronization is challenging because each machine operates with its own clock, and there is no built-in synchronization between them [99]. Consequently, achieving high-precision trace synchronization between different machines becomes a nontrivial task.

Eclipse Trace Compass offers a feature for trace synchronization. It aligns traces from various machines to the same time reference, ensuring they share the same timing. The timestamps for events from the reference trace remain unchanged, but those from the synchronized traces are adjusted based on a formula derived from the synchronization process [34]. The algorithm uses the Convex-Hull method to synchronize clocks in distributed systems by processing packet send-receive pairs from existing network traffic. It identifies pairs with minimum latency to refine synchronization parameters continuously, updating in constant average time (O(1)) per update. This method achieves high precision without adding network traffic, ensuring real-time trace analysis with microsecond accuracy and immediate updates. [100] In our work, we leverage this feature to facilitate our analysis.

4.3.3 Comparison and Visualization

Experts need to compare two sets of executions in distributed systems to identify performance bottlenecks, optimize system configurations, and improve source code efficiency. This comparison helps in pinpointing issues, fine-tuning configurations, ensuring updates are beneficial, detecting anomalies, and guiding development and upgrade priorities. Our visualization module, as shown in Figure 4.2, offers a top-down view of performance differences between corresponding spans in each execution group. The source code for this module is available on GitHub [9].

At the highest level, it displays the differences in execution times for each span. At lower levels, it shows how long each span spends in each system state and details the execution time for each system call.

Flame graphs are created to visualize the performance of spans in distributed systems by illustrating the hierarchy and duration of service calls across different nodes. They help in understanding where time is being spent during execution within the distributed system.

Each flame or box represents a service or span, with its width corresponding to the time consumed. By merging flame graphs from multiple spans within a group, we create a single, comprehensive visual representation. This helps in seeing the overall performance characteristics and identifying patterns or common bottlenecks within the group. Once the flame graphs for each group are merged, comparing them becomes easier. The differences between the two groups can be highlighted more effectively. To facilitate insightful comparison, we devised the *Enhanced Differential Flame Graph*. This unique representation not only highlights disparities between the two groups but also emphasizes the most critical differences using distinct color tints. This makes it easier to pinpoint significant performance variations.

We provide a Filtering and Checklist of selected spans to help users eliminate irrelevant data, ensuring the comparison focuses on the most pertinent information. Creating and merging flame graphs allow us to visualize and compare the performance of different execution groups in distributed systems more effectively. The enhanced differential flame graph further aids in highlighting the key differences, making the comparison clearer and more insightful.

For instance, after a web application update, engineers observe slower response times for user login requests, especially during peak hours. Using the visualization module, they compare login requests from before and after the update. The Differential Flame Graph highlights spans that take longer, such as the SQL manager service, with a red color. Developers then

9. `https://git.eclipse.org/r/c/tracecompass.incubator/org.eclipse.tracecompass.incubator/+/203508`

delve into this specific service to identify the root cause. By utilizing the Metric Extraction module, they discover a significant increase in I/O wait time, with system calls like read() and write() taking longer. To address this, they consider reconfiguring the storage subsystem, optimizing file access patterns, and implementing caching to reduce I/O wait times and improve overall performance.

Filters

The first stage of our implementation involves a filtering process that enables us to narrow our focus to specific groups of interest. Through this essential step, we effectively isolated and identified the relevant requests that served as the foundation for our subsequent comparison. To aid users in this process, we provided them with distribution functions for the two most critical criteria: duration and self-time.

When user selects a portion of the distributed requests, the other section automatically updates to show the relevant requests. This interactive method is shown in Figure 4.2. We present a list of selected requests. This feature allows users to easily remove unwanted requests, simplify the filtering process, and ensure that only the most similar requests contribute to the comparison analysis.

Once the first stage was completed, the merged flame graph of the first group was represented. The same process was used to filter the second group. Users then select relevant requests and lead them to a differential flame graph. This allows for the discovery of relationships between the performance metrics.

Enhanced Differential Flame Graph

Our approach, as discussed in Section 4.2, covers the limitations of previous work, which was limited only to single-thread or single-host execution. It also adapts the differential flame graph to show not only CPU consumption, but also different metrics, revealing latencies caused by waiting threads.

Our solution employs a differential flame graph to compare the execution time of spans and services across multiple executions in distributed systems. Building on the concept of flame graphs, which are presented as icicle graphs, the differential flame graph provides an aggregated view of how execution time is distributed across services and spans derived from call stacks. The purpose of the differential flame graph is to offer a comprehensive visualization, enabling users to quickly identify and understand differences in execution time between two sets of executions. The process of constructing the differential flame graph

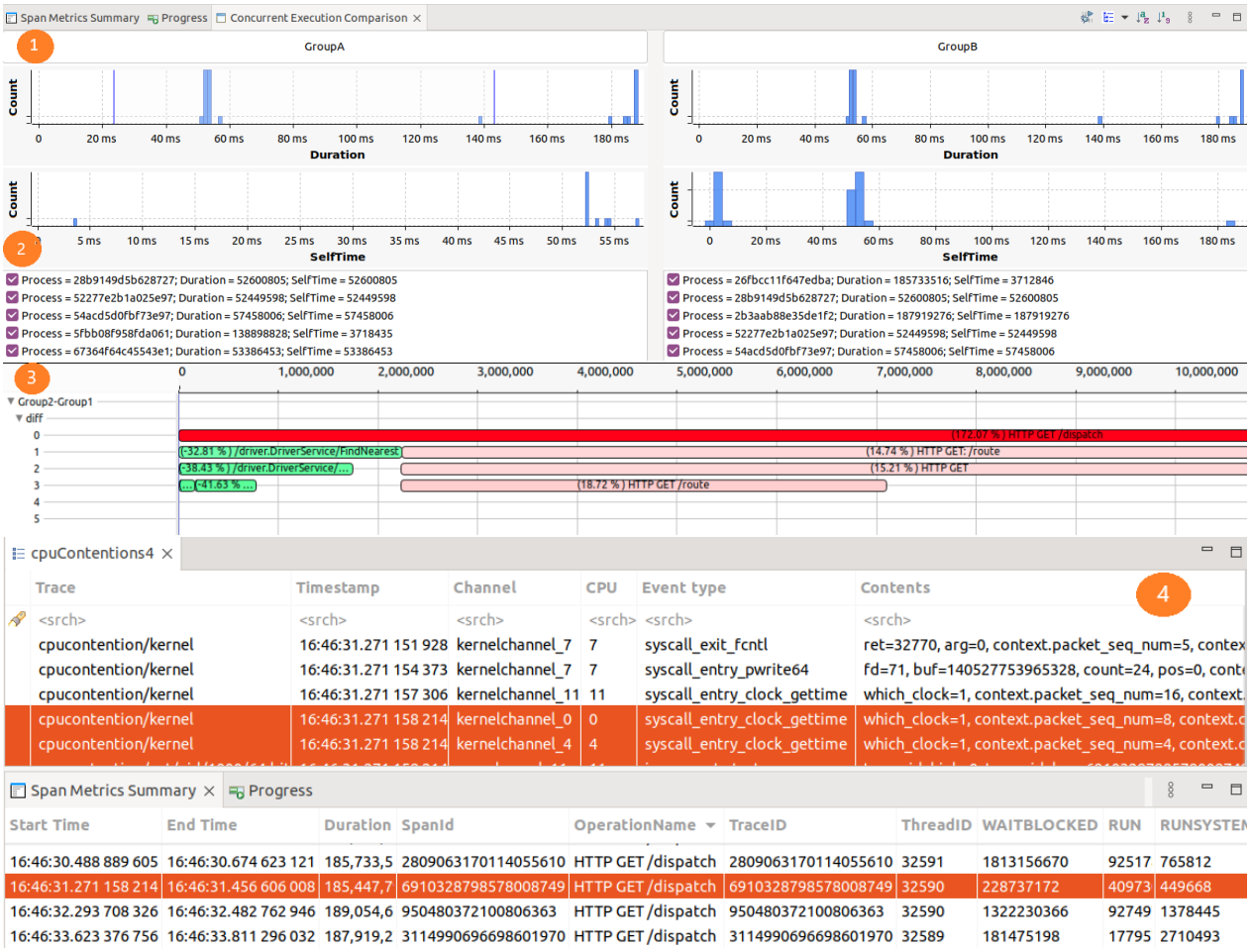


Figure 4.2 DTraComp Visualization and Comparison Module: In the top section, users can filter the requests they will compare in two groups, based on the Duration time and Self-time. In the middle section, users can manually select the requests that they are going to compare. In the bottom section, an Enhanced Differential Flame Graph is presented. Each color has its meaning, explained in more details in Section 4.3.3. At the bottom, we have the Metric Extractor module. This module lists all UST and Kernel metrics along with their specifications. Below that, it displays all spans, including details such as Span ID, Trace ID, Start Time, End Time, and the duration of all thread states related to each span, along with the duration of system calls. When a user selects a span, the corresponding Kernel and UST events are automatically highlighted. The demo is available on YouTube [11]

consists of three main phases, described as follows:

— **Phase 1: Flame graph creation per request**

In this phase, a flame graph is generated for each individual request. This graph aggregates the spans from the call stack for that request, visually representing the execution time of spans and their associated services.

— X-Axis representation: The x-axis of the flame graph represents the total execution time of spans. Unlike a timeline, this axis shows the cumulative time spent in each span, including the time spent in its child spans.

— Span visualization: Each span is represented as a box whose width corresponds to the time spent in that span and its children. This visualization helps pinpoint areas where the system spends the most time, providing insights into performance bottlenecks or areas of improvement.

At the end of this phase, a flame graph is created for each request, forming the foundation for further aggregation and comparison.

— **Phase 2: Aggregated flame graph construction for request groups**

This phase focuses on creating a unified, aggregated flame graph for each group of requests, as defined by the user. Typically, users select requests with similar latencies as one batch in one group to compare with another batch of requests that have shorter or longer latencies. Therefore, the variations of latencies within each group are negligible. The aggregated flame graph represents the average execution times of spans and services across all requests in the group.

— Span merging: For each pair of requests within the group, spans with identical identifiers (e.g., span names) are merged. The duration of these merged spans is calculated by averaging the execution times across the requests: Average span duration = $\frac{t_1+t_2+\cdots+t_n}{n}$ where $t_1, t_2, \ldots, t_n$ are the execution times of the span in each of the n requests within the group.

— Span inclusion: Spans that appear in only one request are directly included in the aggregated flame graph with their original execution time. These spans are preserved as-is since they have no counterpart for averaging in other requests.

This phase results in a single, unified flame graph for each group of requests, which reflects the average performance of spans and services across all requests in that group.

— **Phase 3: Differential flame graph construction**

The final phase involves constructing the differential flame graph by comparing the aggregated flame graphs of two different groups of requests. The goal is to visualize the differences in execution time between the two sets of executions.

— **Differential ratio calculation:** For each span present in both flame graphs, we

calculate the percentage change in execution time:

$$\Delta\%(f) = \frac{t_B(f) - t_A(f)}{t_A(f)} \times 100$$

where $t_A(f)$ is the execution time of span f in Flame Graph A, and $t_B(f)$ is its execution time in Flame Graph B. This calculation highlights whether the execution time of Span increased or decreased between the two groups.

— **New spans:** If a span exists in Group B but not in Group A, it is included in the differential flame graph without a differential ratio (NaN). These newly introduced spans are highlighted, potentially indicating areas responsible for delays or added functionality.

— **Color-Coding Scheme in Differential Flame Graphs:** In our analysis of differential flame graphs, we implemented a color-coding scheme for rectangles based on the evaluated differences between values from two flame graphs. This approach provides a clear visual representation of spans that exhibit significant changes between the graphs. The evaluation is divided into four specific cases:

— NaN Values: When the difference value is NaN, it signifies that a span present in Flame Graph A does not exist in Flame Graph B. In such cases, we assign a dark red color to the corresponding rectangle. This choice of color highlights the complete absence of the span in one of the flame graphs, drawing attention to this significant discrepancy. Mathematically, this is represented as follows:

$$\text{Color} = \text{RED} \quad \text{if} \quad \text{difference} = \text{NaN}$$

— No Difference: For situations where the difference value falls within the range $[-0.05, 0.05]$, indicating minimal or no difference between the two flame graphs. We chose the threshold of 0.05 based on the experiences of our industrial partners, who indicated that differences below this value are not significant. We assign a white color to the rectangle. This color choice reflects that the duration time of spans is nearly identical in both graphs, signifying no meaningful change. The mathematical representation of this condition is:

$$\text{Color} = \text{WHITE} \quad \text{if} \quad -0.05 \leq \text{difference} \leq 0.05$$

— Negative Differences (Shorter Duration): When the difference value is negative, it indicates that the duration time of span in Flame Graph B is shorter compared to Flame Graph A. In this scenario, we assign a green shade to the rectangle. The hue of green becomes darker as the difference value be-

comes more negative, while the intensity of green remains constant. This effect is achieved by scaling the difference value by a factor step, where step = MAX_HUE − MIN_HUE, and then adding this to MAX_HUE (255). The formula for this is:

$$\text{Color} = \text{RGB}(0, 255, \text{MAX\_HUE} + \lfloor \text{difference} \times \text{step} \rfloor) \quad \text{if} \quad \text{difference} < 0$$

For example, with a difference of -0.5 and MAX_HUE = 255, the green intensity is calculated as:

$$\text{Green Intensity} = 255 + (-0.5 \times 255) = 127.5$$

This results in an RGB color value of $(0, 255, 127)$, where the darkening green color visually emphasizes the reduced duration of the span in Flame Graph B.

— Positive Differences (Longer Duration): For positive difference values, indicating that the duration time of span in Flame Graph B is longer compared to Flame Graph A, we assign a red shade to the rectangle. As the difference value increases, the hue of red becomes lighter, while the red intensity remains constant. This is mathematically represented by normalizing the difference value by dividing it by a threshold value fMinThreshold, then scaling it by step, and subtracting the result from MAX_HUE. The formula is:

$$\text{Color} = \text{RGB}(255, \text{MAX\_HUE} - \lfloor \left(\frac{\text{difference}}{\text{fMinThreshold}} \right) \times \text{step} \rfloor, \text{MAX\_HUE} - \lfloor \left(\frac{\text{difference}}{\text{fMinThreshold}} \right) \times \text{step} \rfloor) \quad \text{if} \quad \text{difference} > 0$$

For instance, with a difference of 0.5, MAX_HUE = 255, and fMinThreshold = 1, the green and red intensities are calculated as:

$$\text{Green and Red Intensities} = 255 - (0.5/1) \times 255 = 127.5$$

This produces an RGB color value of $(255, 127, 127)$, where the lightening red color visually emphasizes the increased duration of the span in Flame Graph B.

4.3.4 Metric Extractor

This module helps users identify root causes by offering two levels of system-metric extraction. In the first level, each span is executed by one or more threads, with each thread transitioning through different states during its lifetime, such as Running, Interrupted, Waiting, and others. By calculating the time a thread spends in each state, we can precisely determine how much time a span spends in specific states like Interrupted or Waiting for CPU. This level provides detailed information about system states for each span, including time spent in User-space, System calls, Waited (blocked), Interrupted, Waiting for CPU, and Unknown states.

The second level focuses on system calls executed for each span. System call events can reveal various performance issues. For example, File Operations such as openat, write, and unlink handle file creation, reading, and deletion, which can indicate file I/O bottlenecks. Process Management calls like clone, fork, and execve are responsible for process creation and multitasking, pointing to potential resource allocation problems. Memory Management calls, including mmap, munmap, and brk, manage memory allocation, helping to detect memory leaks or protection errors. Networking and Sockets calls such as socket, bind, and recvfrom are critical for network communication, highlighting network-related performance issues. Signal Handling events like rt_sigaction and kill manage inter-process communication, which can expose communication delays. Timers and Clocks events, such as clock_gettime and nanosleep, manage time, potentially revealing issues with delays or timing accuracy. Finally, Miscellaneous calls like gettid and prctl handle thread management and process control.

This module also highlights all the kernel events related to the selected span, which helps users extract more information, such as the number of kernel events and their sequence, as depicted in Figure 4.3. Users can select the level of detail they need by enabling the collection of the required kernel events during the trace collection phase. More details are explained in Section 4.3.1.

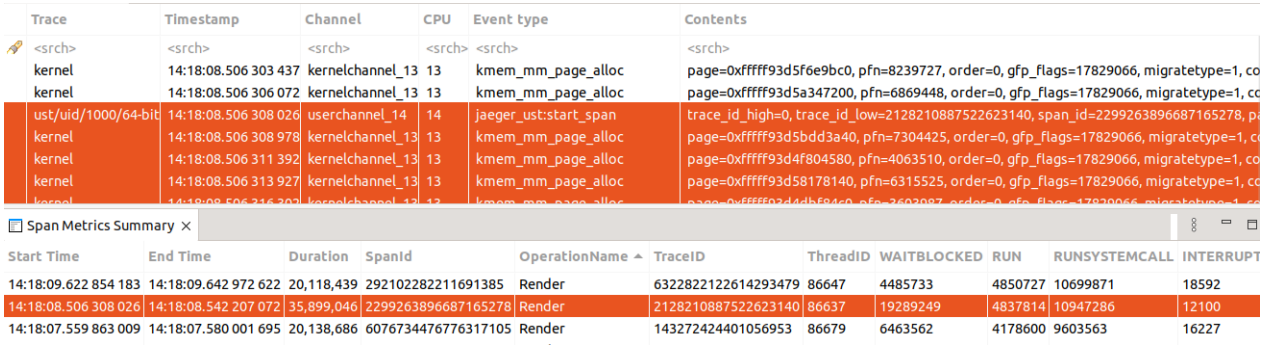
System State Events

Kernel events were required to extract the thread state. Events collected by the trace collector module are used to calculate the status and metrics of each span, such as the Wait Blocked Time, User Space Execution Time, System Call Execution Time, Interrupted Time, Wait for CPU Time and Wait for Fork. In turn, this is useful for a faster understanding of execution time details.

For instance, some executions were waiting for the CPU, whereas others were blocked by a timer. The *sched_switch* event occurs when a currently running thread is being replaced by a new thread, which is ready to begin execution on the same CPU core. This event resulted in two state changes in the system. It changes the state of the new thread to running, and the state of the old thread to either runnable or blocked. Three specific *entry/exit* events signify when a thread enters an interrupted state. These events are *irq_handler*, *softirq*, and *hrtimer_expire*, which correspond to hardware, software, and timer interrupts, respectively. When the event *sched_wakeup* occurs, it indicates that the state of a thread has changed from blocked to runnable, and the context of the event shows the reason for blocking.

For example, if this event is called from a hardware timer context, the thread is blocked by

a timer. Once the thread has been woken up, it waits to be selected by the scheduler to run. However, our work extracts the state of each span, and not simply a thread, because a thread can execute several spans. Thus, by combining the start time of a span and the time of change of a correlated thread, we can determine the state of the span. The output of the work is illustrated in Figure 4.3. The results are in the form of a table that shows the execution time of each span, in all system states, in nanoseconds.



Trace	Timestamp	Channel	CPU	Event type	Contents
<srch>	<srch>	<srch>	<srch>	<srch>	<srch>
kernel	14:18:08.506 303 437	kernelchannel_13	13	kmem_mm_page_alloc	page=0xffffff93d5f6e9bc0, pfn=8239727, order=0, gfp_flags=17829066, migratetype=1, co
kernel	14:18:08.506 306 072	kernelchannel_13	13	kmem_mm_page_alloc	page=0xffffff93d5a347200, pfn=6869448, order=0, gfp_flags=17829066, migratetype=1, cc
ust/uid/1000/64-bit	14:18:08.506 308 026	userchannel_14	14	jaeger_ust:start_span	trace_id_high=0, trace_id_low=2128210887522623140, span_id=2299263896687165278, pa
kernel	14:18:08.506 308 978	kernelchannel_13	13	kmem_mm_page_alloc	page=0xffffff93d5bdd3a40, pfn=7304425, order=0, gfp_flags=17829066, migratetype=1, co
kernel	14:18:08.506 311 392	kernelchannel_13	13	kmem_mm_page_alloc	page=0xffffff93d4f804580, pfn=4063510, order=0, gfp_flags=17829066, migratetype=1, co
kernel	14:18:08.506 313 927	kernelchannel_13	13	kmem_mm_page_alloc	page=0xffffff93d58178140, pfn=6315525, order=0, gfp_flags=17829066, migratetype=1, co

Span Metrics Summary

Start Time	End Time	Duration	SpanId	OperationName	TraceID	ThreadID	WAITBLOCKED	RUN	RUNSYSTEMCALL	INTERRUPT
14:18:09.622 854 183	14:18:09.642 972 622	20,118,439	292102282211691385	Render	6322822122614293479	86647	4485733	4850727	10699871	18592
14:18:08.506 308 026	14:18:08.542 207 072	35,899,046	2299263896687165278	Render	2128210887522623140	86637	19289249	4837814	10947286	12100
14:18:07.559 863 009	14:18:07.580 001 695	20,138,686	6076734476776317105	Render	143272424401056953	86679	6463562	4178600	9603563	16227

Figure 4.3 The Performance Metric Table consists of two sections: The top section displays comprehensive information about kernel and user space events related to all spans. The following section provides an overview of all spans within a trace, with the system state and system call execution time of each span. By selecting an individual span, users can easily access and highlight all the associated kernel and user space events.

System Call Events

System calls establish a crucial interface that enables user-level applications to engage with the foundational operating system kernel, and gain access to privileged resources. Using system call data, detailed system information can be retrieved, including file system operations, process management, memory management, inter-process communication, networking protocols, time and date, and device access. For each of these categories, we have several system call events, such as *open/close*, *read/write*, *mkdir/rmdir*, *rename/unlink*, and *chdir/getcwd*, related to file systems. By measuring how long each of these system call events takes for each span, in the distributed system requests, we can accurately determine how much time a specific span remains in a certain system state. This approach helps us identify the root causes of changes in system performance.

4.4 Case Studies

This section presents five performance problems encountered in real distributed systems [101, 101–107] and describes how our approach can diagnose them. Our framework is de-

signed to identify and address performance issues in diverse distributed systems, regardless of the specific architectural configuration, such as a distributed database, computing clusters, distributed file systems, or distributed web servers. All the case studies were conducted on five real open-source applications (i.e., Eclipse Theia, HotRod, Tidb, JFreeChart, and Apache Cassandra).

The reproducibility of performance problems is a significant challenge; some scientific papers describe performance problems in real applications, but they do not specify the source code [108]. However, most of the bugs reported in open-source repositories, such as GitHub, rarely contain the environmental setup details (e.g., hardware specifications and system configurations) necessary to fully replicate the problem. Some of the distributed performance problems require several containers or machines to reproduce the problem, and more importantly, applying double instrumentation in all of their services is not easy for benchmarking. Therefore, we must consider the real problems in different areas commonly used in distributed systems, with modifications manually introduced to reproduce real problems in the specific version of the mentioned open-source application, or in our own application, for benchmarking purposes.

In this section, we show how DTraComp helps users diagnose some real performance problems in popular real-world distributed applications, including JFreeChart [12], TiDB [13], HotROD [14], Apache Cassandra [15], and Eclipse Theia [16]. We also demonstrate that our software can depict the Enhanced Differential Flame Graph for monolithic applications as well as those with single-level instrumentation.

JFreeChart is a popular Java framework for drawing charts. TiDB is an open-source distributed SQL database that provides high availability, horizontal scalability, and strong consistency across distributed data. HotROD (Rides on Demand) is an open-source demo application that consists of several microservices and illustrates the use of the OpenTracing API. Apache Cassandra is an open-source NoSQL-distributed database. Eclipse Theia is an open-source, extensible platform for building integrated development environments (IDEs) and cloud-based applications.

12. https://www.jfree.org/jfreechart/
13. https://www.pingcap.com/
14. https://github.com/jaegertracing/jaeger/blob/main/examples/hotrod/README.md
15. https://cassandra.apache.org/_/index.html
16. https://theia-ide.org/

4.4.1 Environment

All the experiments were run on two computers with an AMD Ryzen 7 5700G, with a Radeon Graphics CPU running at 1400 MHz, and 32 GB of DDR4 memory. Ubuntu 20.04.1 was used with the Linux kernel version 5.15.0-71, while the LTTng version was 2.13.9-1. All collected traces are available for future replication in the provided traces.

4.4.2 Problem Case 1: Patch Update

Problem Summary

The developers of the Eclipse Theia IDE applied a patch with few changes. The primary purpose of the patch was to improve the startup performance of the Eclipse Theia IDE. Specifically, the patch aimed to reduce startup time [17] [18] [19]. The goal was to make the IDE load faster by optimizing how plug-ins are initialized. By implementing headless plug-ins, the patch aimed to defer the loading of certain plug-ins until they were needed, rather than loading everything upfront. The implementation of headless plug-ins refers to a strategy where plug-ins are loaded in a headless mode, meaning they are initialized and prepared for use without needing to fully activate their user interface components until necessary. This approach helps in improving the performance of an application, particularly in reducing startup times.

Diagnosis

Based on their traces, our differential flame graph in Figure 4.4 shows performance improvements, in most of their services, between the two executions, before and after applying the patch, as illustrated in Figure 4.4. The improvements stem from a patch designed to enhance the startup performance of the Eclipse Theia IDE by reducing startup time. Our framework is fully capable of handling traces from both centralized and distributed systems.

17. https://github.com/eclipse-theia/theia/blob/d34c0b3a8db8483da5b2bf25176e495110b25e52/sample-plugins/sample-namespace/plugin-gotd/README.md.
18. https://github.com/eclipse-theia/theia/blob/master/doc/Plugin-API.md
19. https://eclipsesource.com/blogs/2024/02/09/eclipse-theia-1-46-release-news-and-noteworthy/

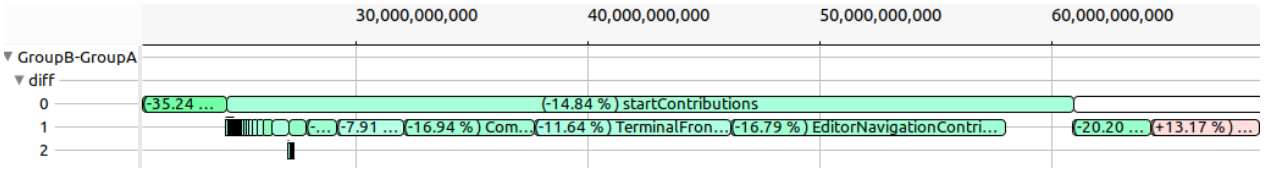


Figure 4.4 Trace of the Eclipse Theia IDE showing performance improvements after applying one patch. By comparing two versions of the application, this framework helps experts in identifying the services that experienced performance improvements and those that did not.

4.4.3 Problem Case 2: Lock Contention

Problem Summary

We found a major problem with Tidb. Adding an index to the github_events table in Tidb (version 7.4.0 - commit 6fbf299) results in a 70% drop in performance. However, no performance degradation was expected. The command before the change ran in an average of 4397 ms; however, with the bad commit (6fbf299), the same command ran in approximately 7663 ms. This performance drop was unexpected and required further investigation to understand the underlying cause, as no significant degradation should have occurred after adding the index. For more information, take a look at this GitHub issue: https://github.com/pingcap/tidb/issues/44584. We ran a mixed read/write workload using Sysbench, which generates concurrent queries, utilizing 32 threads for 5 minutes. This test allowed us to compare the performance differences between the good and bad versions of TiDB to assess the impact of adding an index.

Diagnosis

Inspired by this real performance degradation, our comparison module found an increased execution time for one method between the two versions of Tidb, in Figure 4.5. DTraComp, in Figure 4.5, in its first level of metric extraction, revealed that Waited Blocked increased significantly. Also, the Run system Call increased. The increase in waiting blocked time suggests that there may be increased contention for resources, such as locks or other shared resources, in the bad commit. This contention could cause threads or goroutines to spend more time waiting to access these resources.

The source code of the bad commit revealed that the method acquires a write lock on the mutex before accessing the shared resources in the bad implementation. This effectively blocks other operations (both reads and writes) until the lock is released, which can lead to performance issues and potential contention. Before the change, the method acquired a

read lock on the mutex, before accessing shared resources. This allows multiple concurrent read operations to occur without blocking each other, making it more suitable for scenarios with frequent reads. In the poor implementation, with a write lock, both read and write operations are serialized, leading to contention, increased wait times, and longer execution times. In the good implementation with a read lock, concurrent read operations are allowed, reducing contention and wait times for read-heavy workloads, improving overall performance metrics.

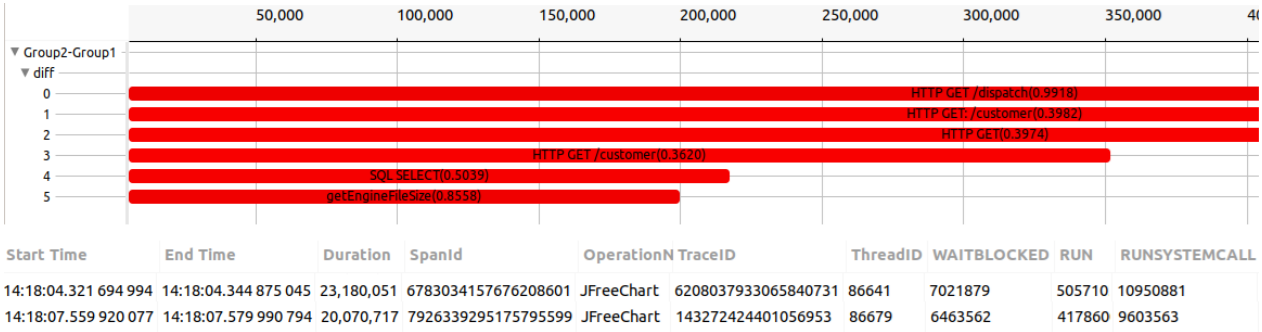


Figure 4.5 Significant regression in Tidb performance problems reported in [2] related to lock contention. The Enhanced differential flame graph shows the latency increase in some services, reaching 99% in the overall latency of a request. The metrics extraction module extracts detailed system information, related to the selected span, « getEngineFileSize » causing a latency propagation to all its related services.

4.4.4 Problem Case 3: CPU Contention

Problem Summary

We have seen that our industrial partners sometimes define as a performance problem any execution not running at the expected speed. In some cases, it is eventually determined that it is simply the result of interference from some other CPU-intensive applications. To replicate this scenario, we conducted an experiment involving an application that performs mathematical operations, and employed the Sysbench benchmarking tool [20] to impose a significant workload on all the available CPU cores. This Sysbench command runs a CPU workload using 16 threads for 5 minutes, fully utilizing the system CPU to simulate high contention conditions. Subsequently, we executed the same application without an additional CPU load. Both executions were performed with identical priorities, on the same CPU cores.

20. `https://github.com/akopytov/sysbench`

Diagnosis

Our visualization module successfully diagnosed the issue by highlighting a significant increase in the time it took for a service that requires CPU intensive mathematical computations, as shown in Figure 4.6. Our metric extraction module revealed that, for this service, both the Waited CPU (time spent waiting for CPU access) and Interrupted (time spent handling interruptions) metrics significantly increased compared to the execution without the additional workload. This indicates that the longer execution times are due to CPU contention, where the system resources are being overwhelmed by concurrent processes.

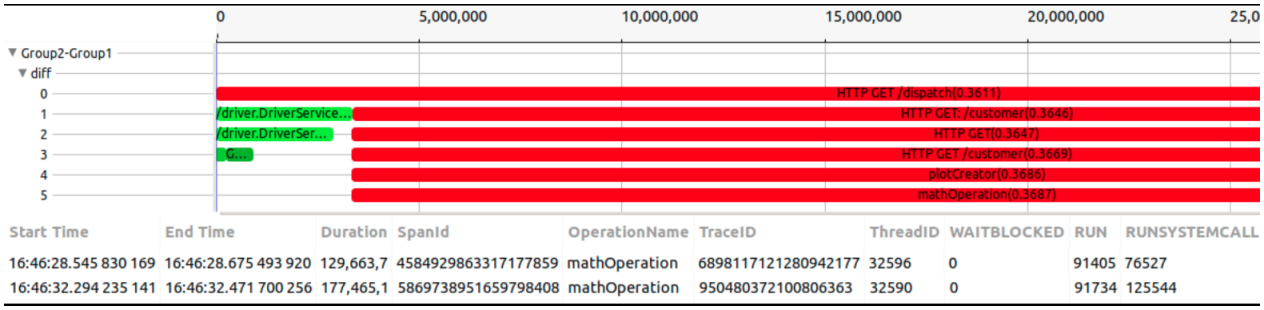


Figure 4.6 The Enhanced Differential Flame graph, alongside the extracted system metrics, can reveal that, for services that have experienced increased CPU wait times, experts can readily infer the presence of a CPU load caused by external applications.

4.4.5 Problem Case 4: Network Overhead and I/O Operations

Problem Summary

The Apache Cassandra database, in versions prior to 2.2, experienced a potential performance problem under certain conditions, particularly in cases where there were frequent or concurrent read requests. The read repair feature in Apache Cassandra is designed to ensure data consistency across replicas during read operations, by detecting and correcting inconsistencies in data. However, this feature experienced high latency during the high read and write query periods.

Diagnosis

Inspired by the performance degradation in the Apache Cassandra database, we simulated a central data store, concurrent client reading, and potentially updated data. We also implemented a basic form of read repair by having clients perform update operations, on the data store, when they read the data. By comparing all the performance metrics for this span, and

the differential flame graph shown in Figure 4.7, we noticed a significant increase in CPU usage, network overhead, and disk I/O.

The metric extractor module illustrates that the improved implementation is faster than the previous version. The Waited blocked metric shows that the resource usage in the improved version is better, and the improvement in « write », « read », and « connect » System calls reveal the efficiency of I/O operations. The « madvise » system calls indicated that memory management was improved in the enhanced version.

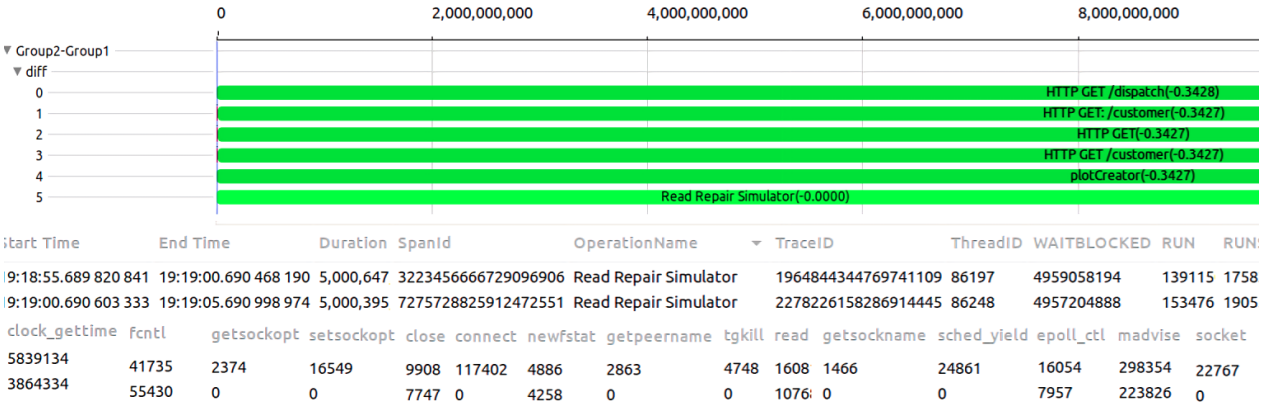


Figure 4.7 The metric extractor module can illustrate the improvements in CPU usage, network overhead, and I/O operations, by providing execution time related to the two versions of the read repair feature.

4.4.6 Problem Case 5: Distributed Deadlock

Since we wanted to demonstrate that our two visualizations can function independently while also complementing each other, for the first use case, we only used the UST-level events and showcased the performance degradation in the differential flame graph. In this use case, we relied only on our metric extraction module.

Problem Summary

As the demand for distributed processing increases, so does the complexity of managing the deadlocks. In distributed databases, the conditions for deadlocks are the same as in centralized databases, but are more challenging to detect, avoid, and prevent [101]. Because this is a common problem in distributed databases, we simulated this performance problem with a distributed database system containing three nodes, each responsible for storing and managing a subset of the data. Applications interact with the database by executing transactions that involve reading and updating the data across nodes. Multiple transactions

from different applications attempt to read and update the data simultaneously. Most of the time, the entire operation required approximately 20 ms. However, a fraction of the time, the operation could not be completed and it was terminated.

Diagnosis

DTraComp helped to identify that the long execution time in some requests came from the abnormal Waited Blocked metric in Figure 4.8. It highlighted the thread responsible for the execution of the long span, and showed which system call increased. We analyzed the code of this span and discovered that the requests attempted to acquire two resources in a specific order. If two requests, on different nodes, hold one resource and wait for the other, a distributed deadlock occurs. In distributed systems, in which deadlocks pose a fundamental challenge, DTraComp can streamline the process of pinpointing issues across diverse nodes.

TraceID	ThreadID	WAITBLOCKED	RUN	RUNSYSTEMCALL	INTERRUPTED	WAITCPU
2860771381546953299	90867	15033596831	815515	183236	1955	8520
3756147765484241812	90862	0	130100	87988	26677	0

Figure 4.8 DTraComp diagnosed the issue through prolonged request execution and an abnormal « Waited Blocked » metric. It identifies the problematic thread and highlights the contributing system calls. Analysis revealed a distributed deadlock caused by resource acquisition order conflicts.

4.5 Evaluation

In this section, we conduct a comprehensive evaluation of our framework from multiple perspectives. To the best of our knowledge, this is the pioneering instance of an open-source application that provides the capability to compare and extract software performance metrics from distributed systems. There is no established ground truth for the comparison of the results for some of our modules. Therefore, our evaluation focused on assessing the overall instrumentation overhead and operation time for various trace sizes, trace types, and system conditions, under external workloads, for each module within our framework. Our framework has been implemented in Java as an incubator within Eclipse Trace Compass for trace analysis and metric extraction. To facilitate synchronization, we instrumented the source code of the Go client of Jaeger by adding LTTng tracepoints. For simplicity in our evaluation, both the Jaeger services and the LTTng daemon were run on the same physical machine. This machine was equipped with an AMD Ryzen 7 5700G processor with Radeon

Graphics running at 1400 MHz and 32 GB of DDR4 memory. The operating system used was Ubuntu 20.04.1 with Linux kernel version 5.15.0-71, and the LTTng version was 2.13.9-1.

4.5.1 Comparison with Existing Tools

To evaluate the effectiveness of DTraComp in a real-world context, we analyzed its features within a microservices-based system. We modified the source code of the HotROD application to introduce performance degradation scenarios using the same environmental setup described in Section 4.1. The source code is available on GitHub [21]. After a system update, developers observed a noticeable reduction in latency and an overall improvement in performance. They wanted to verify whether these enhancements were consistent across all requests and, more importantly, to understand *why* it happened. Diagnosing such performance changes in a distributed system requires end-to-end tracking across multiple services and detailed system-level metrics to identify the root cause.

Jaeger, a widely used distributed tracing solution, can display the latency of each request in Figure 4.9a. While it identifies added or removed services in Figure 4.9b, it lacks a high-level, automated comparison for existing services. As a result, developers must manually select two requests and open separate windows side by side, checking the numerous spans in Figure 4.10, one by one. This manual approach can be time-consuming and only reveals where latency differs, not *why* it differs.

We also look at **VAMP**, an open-source performance visualization tool. A single view automatically compares the high-level latencies requested in Figure4.13, eliminating the need for users to choose a number of requests to be compared manually. However, it does not have root-cause analysis; it provides the user with a set of services for which latency has changed, but does not provide low-level metric capabilities or cross-machine metrics integration. Therefore, in most cases, developers require further profiling tools (e.g., to determine CPU, memory, or IO usage and how these affect the performance changes.)

For in-depth, system-level details, developers typically rely on profiling solutions like **Perf** and **LTTng**, shown in Figure 4.11. However, these tools currently lack the ability to sync different machines on a single time scale while also not being able to capture high-level traces. Due to the machines operating on different clocks, developers must try other applications to sync distributed traces. Additionally, because the trace is from different machines, it becomes very challenging to observe how performance alterations are distributed throughout the entire microservices framework.

21. `https://github.com/maryamekhlasi/HotROD_Performance_Improvement.git`

DTraComp addresses these challenges by integrating (1) automated end-to-end request comparisons, (2) cross-machine data synchronization, and (3) root-cause analysis in a single framework. Figure 4.12a shows how DTraComp compares two sets of requests at once. Unlike Jaeger or VAMP, it syncs high-level trace data with low-level system metrics across different machines. Developers can immediately spot performance changes and investigate the relevant service usage metrics. The *differential flame graph* in Figure 4.12b visually highlights how (and by how much) performance improved in Service SQL SELECT, and reveals that the improvement propagated to related services.

In our example, we found that the pre-update version showed an increase in Waited CPU, Interrupted, and Run system call metrics, as shown in Figure 4.12c. It proves that the service is under heavy load. The higher Waited CPU shows that processes are spending that time waiting for the CPU, most likely because many tasks are running at the same time. The rise in Interrupted indicates more frequent task switching; this occurs when the system overload has to handle a lot of things at once. The increase in Run system calls means that the program is causing the operating system to work harder on various jobs, such as those dealing with file manipulation. Such changes indicate that the SQL SELECT service was laboring more than usual. This kind of inefficiency would likely be caused by problems in the source code, or configuration specific to the pre-update version of SQL SELECT service. This multi-layer diagnosis, done manually with existing tools, required several repeated investigations and typically took much more than an hour. Using this unified view, it was now possible for users to identify the root cause in less than 30 minutes.

Thus, by combining high-level tracing with low-level profiling in one place, DTraComp provides clear advantages over existing solutions: it minimizes manual comparisons, aligns data across machines and services, and automatically highlights both the where and the why of performance changes.

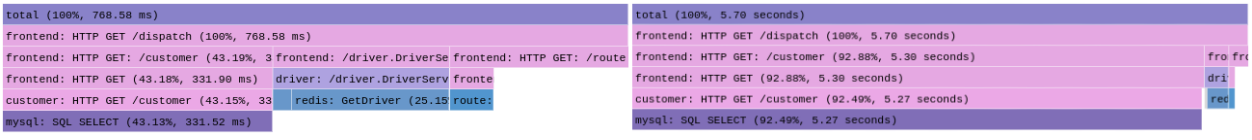
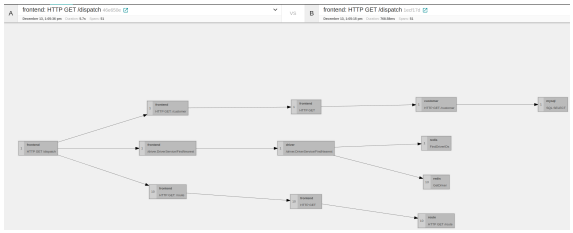


Figure 4.10 An overview of the flame graph displayed in Jaeger for each request. Comparing two flame graphs requires opening them in separate windows and performing a manual comparison.

Table 4.1 compares DTraComp against other open-source tools such as Jaeger, Perf, VAMP, and TraceCompare. This table highlights the unique contributions of DTraComp in performance analysis. While all these tools excel in certain areas, DTraComp fills important gaps

(a) An overview of the Jaeger tracing results, illustrating the latency of each request alongside the span names and counts.



(b) An overview of the comparison feature in Jaeger, indicating added or removed services in red or green but not displaying differences between existing services (spans).

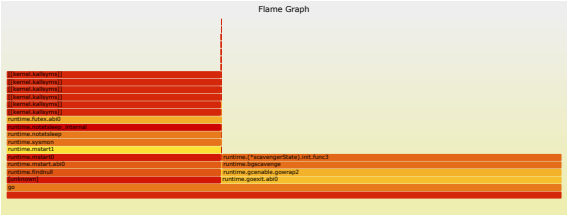
Figure 4.9 An overview of Jaeger views for analyzing end-to-end tracing performance.

by integrating high-level traces with low-level system metrics across distributed systems. It supports automated batch trace comparison, which enables the analysis of several groups of traces at once—a feature not fully supported by other tools. Unlike Jaeger and VAMP, which focus on a manual or high-level-only approach, and Perf or TraceCompare, which are limited to low-level metrics, DTraComp offers a hybrid solution that covers all the critical aspects.

The table also underlines visualization capabilities: The Differential Flame Graph of DTraComp provides full visibility for systemic issues or performance differences, while other tools support either basic flame graphs, as in Jaeger, or differential flame graphs in TraceCompare with limited support for full distributed systems and high-level trace integrations. Furthermore, root cause analysis can be simplified for low effort in DTraComp, which performs the combination of low- and high-level metrics without the need for preprocessing and manual effort.

Another strength is its ability to simplify and speed up the identification of performance issues in microservices compared to tools like Jaeger and Perf. These tools are useful but have limits in ease of use and in analyzing complex systems when latency affects multiple services. As shown in the last row of Table 4.1, DTraComp provides low-effort root cause analysis with both high-level and low-level metrics, which helps find problems faster and more effectively than other tools.

These features make DTraComp a comprehensive and robust tool for diagnosing performance bottlenecks in complex distributed systems, offering solutions that outperform the capabilities of existing tools in both research and practical applications.



(a) An overview of the flame graph by Brendan Gregg, created from kernel events extracted using perf.

```
go 228761 603280.107072:          1 cycles:
        ffffffff8549c061 [unknown] ([kernel.kallsyms])
        ffffffff85301a41 [unknown] ([kernel.kallsyms])
        ffffffff85f9a345 [unknown] ([kernel.kallsyms])
        ffffffff85f9a969 [unknown] ([kernel.kallsyms])
        ffffffff853924b8 [unknown] ([kernel.kallsyms])
        ffffffff85393365 [unknown] ([kernel.kallsyms])
        ffffffff853951ca [unknown] ([kernel.kallsyms])
        ffffffff85395c8b [unknown] ([kernel.kallsyms])
        ffffffff85205c92 [unknown] ([kernel.kallsyms])
        ffffffff85f8c194 [unknown] ([kernel.kallsyms])
        ffffffff86000124 [unknown] ([kernel.kallsyms])
                478103 runtime.futex.abi0+0x23 (/usr/local/go/bin/go)
                410493 runtime.notetsleep_internal+0xb3 (/usr/local/go/bin/go)
                4105c9 runtime.notetsleep+0x29 (/usr/local/go/bin/go)
                44b706 runtime.sysmon+0x1c6 (/usr/local/go/bin/go)
                441f93 runtime.mstart1+0x93 (/usr/local/go/bin/go)
                441ed6 runtime.mstart0+0x76 (/usr/local/go/bin/go)
                4743c5 runtime.mstart.abi0+0x5 (/usr/local/go/bin/go)
                4320fd runtime.(*pageAlloc).alloc+0x15d (/usr/local/go/bin/go)
            c0000120a0 [unknown] ([unknown])
                     0 [unknown] ([unknown])
          7ffea88f5a30 [unknown] ([unknown])
```

(b) An overview of the Linux kernel events captured by perf for performance analysis.

Figure 4.11 A combined overview of capturing Linux kernel events with perf for performance analysis, and the resulting flame graph by Brendan Gregg.

It is important to note that the comparisons presented in Table 4.1 are for informational purposes. They provide context and highlight the features of DTraComp relative to other tools.

4.5.2 Cost of Trace collecting

We evaluate the overhead associated with our instrumentation at each level. In line with previous research [83], which reported a 5% overhead for trace collection using LTTng and Opentracing, we reassessed the overhead of our trace collection based on their approach. The evaluation consisted of four scenarios. We assess the overhead introduced by our instrumentation at both levels of our metric extraction module. The first level focuses on a subset of kernel events to capture the duration of each thread state within a span. The second level goes further, collecting detailed information about system call durations. While gathering data on the large number of system calls can be resource intensive, this information is essential for accurately diagnosing the root causes of performance issues.

The overhead is tested on the HotROD demo application provided by Jaeger, which is already instrumented with distributed tracing using Jaeger. It allows users to order virtual car rides and includes multiple services such as a web front end, a Redis server, and a MySQL database. We simulate a high volume of concurrent requests using the Apache ab tool, where 10,000 total requests are performed, with 10 concurrent requests at a time, and we measure the average request processing time. We had an average of 51 spans per request and also used the default configuration of OpenTracing and Jaeger.

Table 4.1 Feature comparison of DTraComp with related open-source tools. It highlights the scientific contributions, practical features, and strengths in analyzing performance in distributed systems.

Feature	**DTraComp**	**Jaeger**	**Perf**	**VAMP**	**TraceCompare**
Scientific Contributions					
Batch Trace Comparison	**Yes**	No	No	Yes	Yes
Granularity of Analysis	**High + Kernel-Level**	High-Level Only	Kernel-Level Only	High-Level Only	Kernel-Level Only
Practical Features (For Context)					
Open-Source	**Yes**	Yes	Yes	Yes	Yes
Scalability in Production	**Yes**	Yes	Yes	No	No
Compatibility with Multiple Trace Standards	**Yes**	Limited	No	Limited	Limited
Trace Pre-Processing Required	**No**	No	No	Yes	No
Visualization Capabilities					
Type of View	Differential-Flame-Graph Flame Graph Tables Histogram Chart	Flame Graph Timeline	Tables Bar Charts	Histogram Charts Tree diagram	Differential-Flame-Graph Histogram Chart
Effort Needed for Root Causes	**Low, Automatic**	High, Manual, limited to high-level infos	High, Manual, limited to low-level metrics	Medium, Automatic, limited to high-level infos	Medium, Automatic, limited to low-level metrics

— First scenario (No tracing), both Opentracing and LTTng instrumentation were turned off and no traces are collected.

— Second scenario, we enabled Opentracing tracepoints and disabled LTTng, which was our baseline in this evaluation.

— Third Scenario, Opentracing is enabled, and LTTng is in full snapshot mode for the collection of a subset of kernel events required for or analysis in the Metric extraction module, alongside UST events related to our application. LTTng is running in snapshot mode, where events are stored in memory and only saved to a trace file when needed. This improves performance and keeps trace files smaller by recording only when an issue occurs. In our scenario, we ensure that no trace flush is triggered, the events are not written to disk, they are continuously replaced by new ones when the circular buffer fills up.

— Fourth, we enabled Opentracing with the LTTng standard mode, with the same subset

of kernel events as in the previous scenario, and UST events. LTTng operates in standard mode, where all collected events are saved directly to a trace file.

For all the experiments, the LTTng daemon, Jaeger services, and the application, are run on a single machine to simplify the evaluation. The specifications of the machine on which we conducted our test is mentioned in Section 4.5.

Table 4.2 The overhead of our trace collection solution for HotROD, using only the necessary kernel subset and excluding system calls, remains minimal. Compared to the baseline, the overhead is less than 9% in LTTng snapshot mode and approximately 10% in standard mode.

Scenarios	Second (Baseline)	Third	Fourth
First	14.85%	22.59%	23.37%
Second (Baseline)		9.08%	10.07%
Third			1.016%
Fourth			

Table 4.3 The overhead of our trace collection solution for HotROD, which includes the full kernel subset and system calls, is moderate. Compared to the baseline, the overhead is below 26% in LTTng snapshot mode and around 27% in standard mode.

Scenarios	Second (Baseline)	Third	Fourth
First	18.94%	40.13%	41.63%
Second (Baseline)		26.14%	27.99%
Third			2.5%
Fourth			

The results obtained, as shown in Table 4.4, align with the findings mentioned in the referenced article. The trace collection overhead in our default mode for the distributed application had a minor impact on the throughput of the requests. This suggests that the additional tracing introduced by Opentracing and LTTng does not significantly hinder the overall performance of the system. The overhead of extracting more kernel information is depicted in Tables 4.2 and 4.3; however, it results in a higher tracing overhead compared to the default mode. Careful consideration of the tracing environment is necessary to balance the desired level of detail with the associated overhead.

4.5.3 Cost of the Metric Extraction

We used the traces collected from Section 4.5.2, where all the details are thoroughly explained. Our set of traces includes multiple services and spans, each with varying trace sizes both in

Table 4.4 The average execution time of trace collection for 10,000 requests using 10 client threads across the four scenarios explained in Section 4.5.2 and two levels of trace collection modes are available in the proposed tool. The first level, which captures kernel events without activating the system calls, offers a broad overview of the performance degradation of the underlying causes. By contrast, the second level provides precise and detailed performance metrics for each request.

Scenarios	Average time per Request (ms)	
	with syscalls	without syscalls
First	5.551	5.600
Second (Baseline)	13.804	11.699
Third	16.867	11.849
Fourth	16.793	11.684

the volume of requests and the variety of trace types as shown in Table 4.5. As explained in Section 4.5.2, in reference to Table 4.4, the kernel trace is optional but includes voluminous details that help to understand the root cause at the lowest level. We then ran several analyses on each trace and recorded the average execution time for extracting performance metrics through our Metric Extraction module. The process calculates the duration of different thread states for each span, as well as the execution time of various system calls within each span. The results are shown in Fig. 4.14. By extracting all the performance metrics for each of our traces, it appears that our operating time grows consistently with the trace size, and the time it takes for execution is quite reasonable, even for relatively large traces.

We also measured the precision of our evaluation by matching the duration of each span, and the total execution time of our randomly selected requests, with the execution time evaluated by Jaeger, as a baseline for evaluating the accuracy of the total duration evaluated for each span, as shown in Table 4.6. The numbers validate that we can reproduce the results with our implementation by separating the execution time for each system state. OpenTracing was developed using microsecond units. Although this suffices for complex high-level distributed requests, it becomes limited when we need to examine the individual intervals within those requests, to identify the sources of latency. Our results offer greater precision by reporting the execution times in nanoseconds.

4.5.4 Cost of the Comparison

To evaluate the effectiveness of both our Filtering module and Comparison and Visualization module, we conducted a series of experiments. We used the traces collected in Section 4.5.2 to measure the execution times of these two modules. We evaluated the average execution

Table 4.5 Summary of trace size, number of requests, spans, and contributions of kernel and user-space trace (UST) to the total trace size.

Trace Size	Req.	Spans	UST	Kernel
29.6 MB	50	250	428 KB	29.2 MB
89.3 MB	250	1,250	1.3 MB	88.0 MB
326.4 MB	500	2,500	3.0 MB	323.3 MB
978.6 MB	3,000	150,000	17.5 MB	961 MB
1.7 GB	5,000	250,000	29.2 MB	1.7 GB

time of the filtering module as it loads traces and generates bar charts based on both their duration time and self time, and lists all requests for manual selection by the user. We also evaluate the average execution time for generating a differential flame graph. This process is initiated when the user selects two groups for comparison. The results, shown in Figure 4.15, indicate that the execution times of these modules are minimal. However, slight increases in execution times can occur when the trace size or system workload increases. A decrease in the filtering module execution time, for traces with a size of 794 MB, is because this trace only contains two identical requests, therefore our merging module automatically merges these two requests for comparison. Since our comparison and visualization module can separately be used for concurrent traces, compatible with the Eclipse Trace Compass tool, without any need for double instrumentation, and with a time complexity of O(n), it is therefore quite efficient for comparing requests in large industrial traces.

4.6 Conclusion and Future Work

This paper introduces DTraComp, an open-source framework compatible with various microservice trace standards. Unlike existing performance comparison tools, our approach has the unique capability to compare groups of requests from the highest user level down to kernel system events, while also pinpointing the exact reason for performance degradation. Our framework automatically identifies the latency differences between similar requests in distributed systems and incorporates a powerful comparison feature. This feature allows users to compare two groups of identical requests, thereby providing valuable insights into their performance characteristics throughout the distributed system. In addition, the framework offers two levels of detailed kernel-level system information, for each span within a request, enabling a comprehensive analysis of performance degradation. Furthermore, within our framework, various computing configurations are supported, including distributed systems that accommodate both concurrent and sequential services, in addition to monolithic appli-

Table 4.6 Precision of execution time for each span

Service Name	Span ID	Jaeger Duration Time (ms)	DTraComp Duration Time (us)
HTTP GET /customer	4dce9ba044b45859	313,897	314,206,223
DriverService/FindNearest	118142188d13609c	216,620	216,871,328
HTTP GET /route	7fbc78872f3847df	41,743	42,017,778
HTTP GET /route	068be711cbee624e	69,695	69,698,931
HTTP GET /route	69c577822cda2908	49,599	49,572,453
HTTP GET /route	3ef14dfe688096be	42,478	42,496,332
HTTP GET /route	2c1cdfb1683d2262	31,479	31,481,573
HTTP GET /route	64bcebf453d0c6ba	49,609	49,696,770
HTTP GET /route	4d4daf70092f9c01	56,357	56,362,504
HTTP GET /route	5f246b46290f6047	61,724	61,730,655
HTTP GET /route	525a1dc1a5a0a1d5	61,106	61,136,932
HTTP GET /route	602e0b2dad7af025	69,575	69,626,575
HTTP GET: /customer	1c0fd57703da8e0a	315,612	315,548,823
HTTP GET: /route	6d83aaa2ec25eaac	43,402	43,378,498
HTTP GET	6d84d72745b448b3	43,296	43,323,463
HTTP GET	77c0315b649c18e9	315,498	315,536,842
DriverService/FindNearest	43d04e7f1c875ca7	219,060	219,018,811
HTTP GET	1276195a9087245f	50,632	50,653,248
HTTP GET	303f97953df6617a	70,916	70,935,682
HTTP GET: /route	66779dc0155fe817	50,729	70,935,682
HTTP GET: /route	0e2ec848b9b49eb9	71,013	70,979,892
HTTP GET: /route	6e63d5984563fa30	32,351	32,338,806
HTTP GET	069ba949a15aaffe	62,592	62,589,634
HTTP GET	017f9ea524d47e16	32,301	32,298,229
HTTP GET: /route	5cb3a5e63d45112a	62,649	62,634,263
HTTP GET	706546ce1f5de042	70,385	70,411,732
HTTP GET: /route	7926e14d6866f62b	70,491	57,070,487

cations.

In this article, we demonstrate the efficacy and effectiveness of our framework in five different real use cases, showing that our tool can locate several performance problems. Our tool imposes minimal overhead, with less than 9% for optimized kernel subsets and under 27% for full subsets. It scales linearly with trace size. The costs associated with our Metric Extractor, Filtering, Comparison, and Visualization modules remain reasonable, even when processing large trace sizes, ensuring reliability for production deployment. As part of our future work, we plan to extend our filtering module to provide users with more options for selecting and comparing two groups of requests. Additionally, we aim to incorporate anomaly detection methods to gain insights into abnormal traces and facilitate their comparison. Furthermore,

future researchers can work to automatically group and categorize performance problems based on kernel metrics and system state duration times, leading to more efficient analysis and diagnosis.

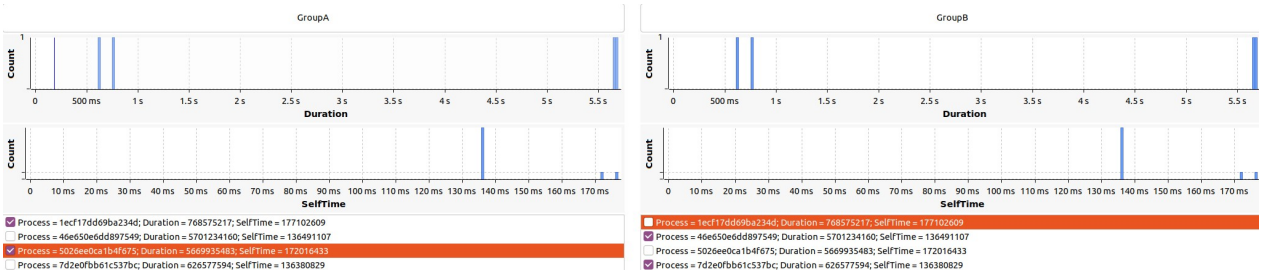
4.7 Threats To Validity

One possible threat to validity could be that clock drift across machines may mismatch timestamps and create errors in the trace synchronization. As our visualization tool is integrated with Trace Compass, it handles this problem using TCP packets for synchronization. Timestamps were adjusted by Trace Compass according to the shared events on the machines so they could be aligned.

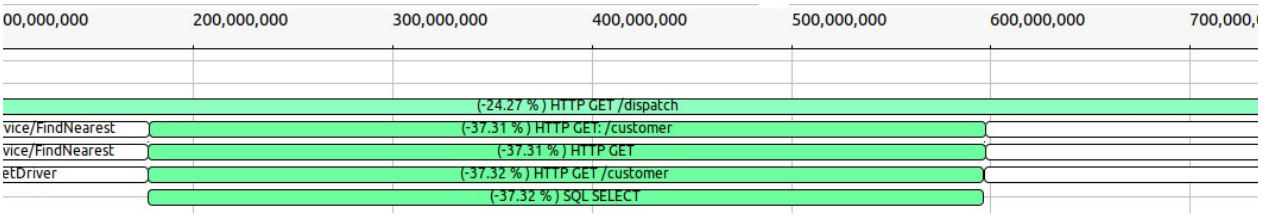
Additionally, it may be necessary to exactly replicate the setups in order to achieve similar results in another environment. Differences in hardware, software configurations, or network conditions could all affect the behavior of the system and the performance metrics. This includes the detailing of the experimental setups, describing modifications made to applications, in addition to providing source code and configurations in a public repository to ensure reproducibility.

4.8 Acknowledgement

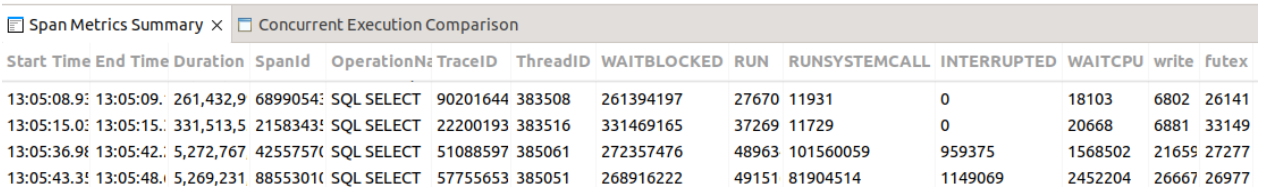
We would like to thank AMD, Ciena, EfficiOS, Ericsson, Natural Sciences and Engineering Research Council of Canada, and Prompt for funding parts of this work.



(a) An overview of the filtering module, showing four total requests and enabling users to manually select requests based on self-time or latency.



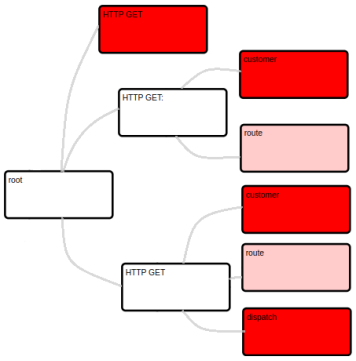
(b) An overview of the differential flame graph generated from requests selected by the filtering module, highlighting improvements for each span and their hierarchical relationships. The graph indicates an overall 37.32% improvement in SQL SELECT operations, with benefits propagating to other related services.



Start Time	End Time	Duration	SpanId	OperationNa	TraceID	ThreadID	WAITBLOCKED	RUN	RUNSYSTEMCALL	INTERRUPTED	WAITCPU	write	futex
13:05:08.9	13:05:09.	261,432,9	6899054	SQL SELECT	90201644	383508	261394197	27670	11931	0	18103	6802	26141
13:05:15.0	13:05:15.	331,513,5	2158343	SQL SELECT	22200193	383516	331469165	37269	11729	0	20668	6881	33149
13:05:36.9	13:05:42.	5,272,767	4255757	SQL SELECT	51088597	385061	272357476	48963	101560059	959375	1568502	21659	27277
13:05:43.3	13:05:48.	5,269,231	8855301	SQL SELECT	57755653	385051	268916222	49151	81904514	1149069	2452204	26667	26977

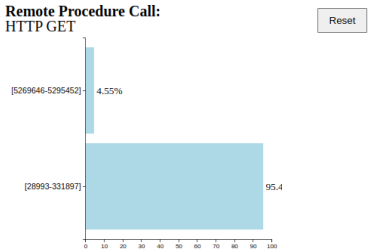
(c) An overview of the metric extractor module, showing detailed system performance metrics for each service. Comparing the WAITBLOCKED and WAITCPU metrics reveals shorter waiting times, meaning there is less competition for resources and smoother operation. This improvement has directly led to faster SQL SELECT operations, especially in the post-update version.

Figure 4.12 An integrated overview of the filtering module request selection, the differential flame graph revealing a 37.32 percent improvement in SQL SELECT operations, and the metric extractor module detailing CPU utilization gains that lead to reduced latency and better overall performance.



(a) An overview of the tree diagram of VAMP, illustrating the hierarchical relationships between services and highlighting the differences in latencies.

(b) An overview of the filtering module, enabling users to select a specific range for viewing the tree diagram.



(c) An overview of a bar chart showing the percentage of requests, with the latency ranges displayed on the y-axis.

Figure 4.13 A general overview of the VAMP performance analysis visualization tool, used to reach an understanding of the system performance by the visualization of service hierarchies, latency distribution, and filtering for targeting analysis.

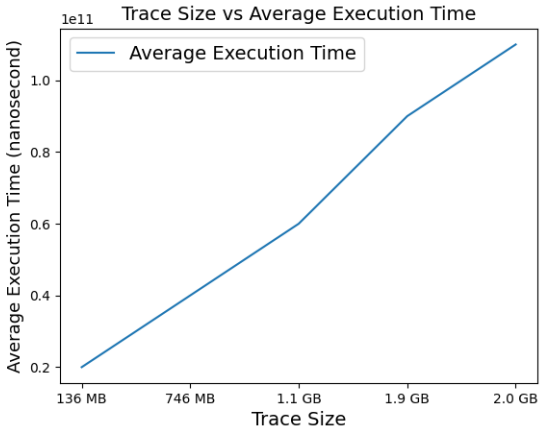
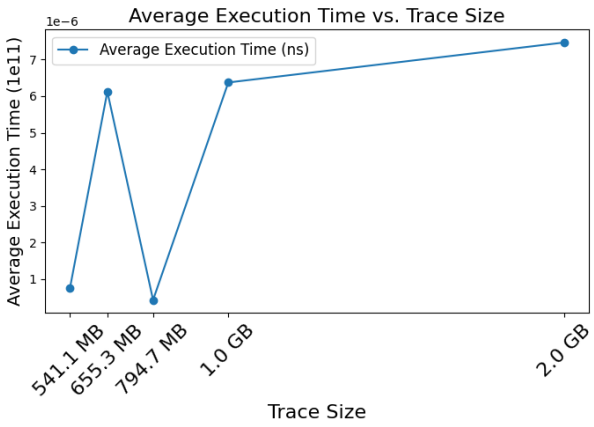
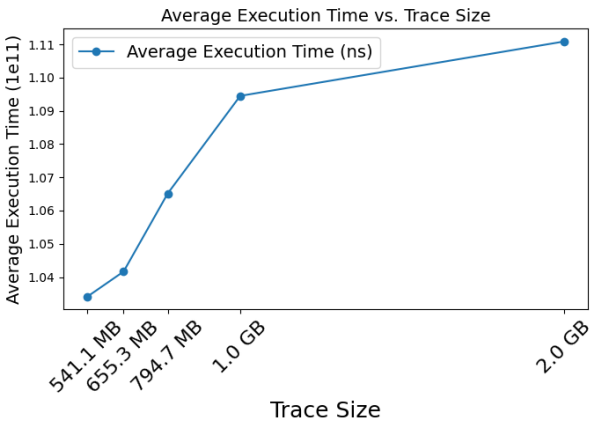


Figure 4.14 Average execution time of DTraComp metric extraction analysis. By increasing the size of the trace, the execution time increased linearly. The time taken for execution is quite acceptable for fairly large traces.



(a) Average execution time of the Merging module in nanoseconds across various trace sizes/-types.



(b) Average execution time of the Filtering module in nanoseconds across various trace sizes/-types.

Figure 4.15 Cost of the Merging and Filtering modules.

CHAPITRE 5 ARTICLE 2: HYBRIDRCA: LIGHTWEIGHT CRITICAL-PATH-AWARE HYBRID TRACING FOR ROOT-CAUSE ANALYSIS IN PRODUCTION MICROSERVICES

Authors Maryam Ekhlasi, Arnaud Fiorini, Michel Dagenais, Naser Ezzati-jivan, Maxime Lamothe

Published at 41st International Conference on Software Maintenance and Evolution 2025. (30 June 2025)

Abstract [Background] Distributed software systems are widely used because they are easily scalable, independent of the programming language, and each service can be autonomous, which makes it easy to update. However, distributed systems have their challenges, particularly in monitoring and maintenance. Since they consist of a large number of services with complex interactions, detecting issues or performance degradations becomes challenging and time-consuming. If a single service is facing a problem, it can propagate to other services, affect user experience, and even lead to significant costs in the industry.

[Problem] End-to-end tracing allows tracking requests within a distributed system, whereby engineers can have an overview of latency in services and their interactions, but they are not able to find the cause of the performance issues. Profiling and system metrics enable engineers to obtain more accurate and detailed system information to identify the reason behind performance changes, but they cannot depict interactions between distributed services. Hybrid approaches can address these limitations but typically require a lot of storage space for high-level and low-level metrics and are generally computationally expensive.

[Method] To find performance problems in distributed systems, we propose an adaptive approach that moves from high- to low-level information to first identify a suspicious service and then determine the reason for the performance degradation. In this approach, we retrieve system details only for the selected services, thereby overcoming some of the key industry issues, such as storage capacity limitations and computing overhead, while also identifying the core reason behind the performance problems.

[Conclusion] We evaluate our method on three microservice systems: HotRod, TrainTicket, and OnlineBoutique, and compare it with state-of-the-art approaches. Our results show that while our method achieves comparable recall and precision to the best existing techniques, it significantly reduces the number of spans to analyze by 21% and storage usage by more than 99%. This makes our approach highly efficient and lightweight while maintaining accuracy

for root cause analysis in distributed systems under resource constraints.

5.1 Introduction

Modern software systems are becoming more complex because of their complicated architectures and the large number of services running on different nodes. When a performance problem happens in one service, it can increase the overall latency of a request, leading to an unsatisfactory user experience [109]. Finding the problematic service and manually understanding the root cause is challenging, time-consuming, and prone to human error because of the complexity; it may take several hours for an expert to identify the issue. [109]. Experts need to know the architecture of the system, the latency thresholds for normal requests, and how to detect performance issues. Therefore, tracing systems is essential for the system reliability [110–112]. Tracing can serve different ranges of information from high-level to low-level, including hardware failures, system configuration, and inter-service communication in production environments. They can also be useful for detecting anomalies [47, 49, 113], predicting failures [114], and performing root-cause analyses [82].

Distributed tracing, or distributed request tracing, collects trace data to present the entire journey of a request across multiple services and demonstrate how each service contributes to overall system latency performance tuning, live debugging, and identifying root cause problems. It highlights how each service contributes to overall system latency [115].

In general, tracing can be classified at different levels. High-level tracing shows the overall flow of requests across services, illustrating how a request moves through the system. However, it does not have many system-level insights, therefore, making it difficult to understand the root cause of performance problems.

Low-level tracing captures fine-grained data, including CPU, memory, and I/O usage, but does not show the full end-to-end flow between services. It also typically requires large memory and storage to keep detailed information (e.g., Netflix exposes two million metrics, and Uber exposes 500 million metrics [40, 116]).

Some research [85] focuses on using high-level traces. Their approaches are useful for finding general performance problems, but they do not explain the reasons behind these problems. This is because they do not analyze detailed system behavior.

Other approaches [117–119] use a hybrid tracing—a combination of high-level and system-level tracing. This requires storing a large amount of trace data and leads to storage and processing overhead—one of the main challenges in the industry. Moreover, since in production the services run on different machines, pods, or containers, there is a need for high-

and low-level trace synchronization, which would ultimately depend on a very accurate clock synchronization among the different components, which is another challenge and further complicates the trace analysis.

In distributed systems, almost all requests will trigger a chain of service calls across many components. The interactions can be viewed as a directed acyclic graph (DAG), in which each service is a node, and each edge shows a parent-child relationship between calls. Causal analysis approaches have been a major focus in understanding how service delays propagate. Although this approach is effective, it typically requires the tracking of a huge number of services with several attributes for each service. All of this becomes computationally expensive for large-scale systems.

Another limitation is assuming that all services are equally important when analyzing performance. However, due to parallelism, multiple child services of the same parent can run at the same time. This means not all services contribute equally to the total request latency. Only the services on the critical path—the longest chain of dependent service calls—directly affect the overall latency and, in turn, the user experience. Failing to account for this can lead to misleading analysis results and incorrect identification of performance bottlenecks.

In this paper, we present an adaptive approach for identifying services that affect the overall latency of distributed requests and negatively impact user experience. All the source codes are available at GitHub [1]. Our approach focuses on answering a key question: What is causing the latency increase, and why is it happening? Our main contributions are as follows:

— We group similar or identical traces for easy and fair comparison to identify abnormal traces.
— We extract the critical path of service calls, which allows us to handle parallel calls by focusing on the services that contribute to the overall latency.
— Only related system-level metrics are collected to keep low memory and storage consumption while still providing relevant details for problem diagnosis.
— We employ a hybrid tracing approach that unifies high-level and low-level traces with clock synchronization in a distributed system.
— Our approach does not require knowing in advance the abnormal time range when the latency increases, which makes it more robust and practical in real-world scenarios.

1. `https://github.com/maryamekhlasi/adaptive_rootCause_analysis/tree/master`

5.2 Background and related work

In this section, we provide background information concerning the fundamental concepts discussed in the paper, and we discuss related state-of-the-art research.

5.2.1 Root Cause Analysis (RCA)

Root Cause Analysis (RCA) is an important process used to find the main reason behind a system failure or performance issue, especially in distributed systems [120, 121]. Many approaches have been developed to automatically identify the root cause of anomalies in recent years [85, 122–128]. Some [85] use service-level to locate performance issues, others [119] rely on system metrics, and some [128, 129] combine both (multi-modality) for better accuracy. Each of these approaches has its own strengths and limitations, which we discuss in the following sections.

5.2.2 High-level Tracing

End-to-end tracing, or distributed tracing, is a method of monitoring and examining the flow of requests across the various components (services) of a distributed system at a high level. This method represents requests as traces that are composed of smaller units called spans, where each span represents an operation or part of the processing. These spans are connected through a unique trace ID that maintains a clear parent-child relationship, and interactions between services. Such a detailed structure shows the entire journey of a request from the beginning to the end and identifies bottlenecks, latency issues, and faults [130].

Some previous studies [85, 131, 132] have utilized high-level tracing approaches to reveal performance issues in distributed systems. These methods typically use aggregated metrics, such as the duration time of each service, to identify suspicious services. Although it shows which services were suffering from performance degradation, it does not have detailed information for root cause analysis. Specifically, these works focus on total duration time without considering self-time, the time a service spends executing its own logic, excluding time spent waiting on downstream services [3]. This makes it difficult to identify the actual source of the problem since latency introduced in one span can propagate to others—a delay in one span can increase the total latency of its parent and even other related spans. As a result, a span that appears slow due to increased total duration may actually be affected by delays in its upstream or downstream dependencies. This kind of error propagation can lead to incorrect conclusions about the true origin of the latency.

5.2.3 Low-level Tracing

Low-level tracing is a method to analyze performance problems faced by computer systems. It relies upon detailed information provided at the levels of hardware and the kernel: CPU usage, memory usage, network throughput, disk activity, and I/O operations. This method traces basic system events like system calls, interrupts, context switches, and kernel activities. Low-level tracing techniques use kernel-based tools such as LTTng [2] and Perf [3] to pinpoint performance problems in computer systems [3, 83]. A few methods [72, 118, 133] merge high-level traces with low-level system metrics to pinpoint performance issues. These methods, however, have two main limitations. First, the performance metrics and high-level traces are collected from different machines, each with its own clock. Hence, it poses issues regarding time synchronization and very hard to align system metrics onto specific spans. Second, these methods generally work at the pod or node level, not at the span level. Therefore, they can only tell which pod is affected by problems, such as CPU contention, but do not really show specifically which spans are impacted due to performance degradation.

5.2.4 Hybrid Tracing Approaches

Some approaches [3, 83] have used hybrid tracing by double instrumentation on the code. This means it collects high-level traces and low-level system metrics for each span. This also resolves the issue of trace synchronization. However, they need to collect kernel metrics for all spans. Kernel-level data contains low-level events such as context switches and system call delays. Since kernel-level data includes low-level events like context switches and system call delays, collecting it for every span can lead to high storage usage, which becomes a challenge in large-scale systems.

5.2.5 Critical Path

In distributed systems, many services run in parallel and contribute not equally to the total request duration. Some spans execute at the same time, so their delays might not directly increase the overall latency. However, earlier work [85, 118, 134] considers all spans to have the same effect on the total duration. This neglects the issue of parallel execution, which can lead to incorrect assumptions about which spans actually affect performance degradation.

Another challenge is the large number of spans in real-world distributed systems. A single request may involve hundreds of spans, especially in microservice-based architectures.

2. urlhttps://lttng.org/blog/2019/07/16/debugging-python-applications/
3. urlhttps://perfwiki.github.io/main/

Analyzing every span and collecting detailed system metrics for each one (such as CPU usage, memory, or I/O) leads to high overhead and large storage requirements. These two factors—parallel execution and the high volume of spans—make accurate and efficient performance analysis more difficult.

The critical path in distributed systems is the longest dependent operation chain in the execution, affecting the total latency of a request. It indicates the slowest path that a request takes during processing; critical path analysis identifies the part to optimize within the system to make it more efficient and to reduce latency [135]. Although many spans also run in parallel due to parallelism, only the spans on the critical path directly impact the overall time of the request.

By focusing only on the spans from the critical path, we can ignore other spans running in parallel without affecting the overall response time. This enables analysis of those spans that contribute to user experience. It also helps reduce the number of spans to examine and the number of system metrics to collect. Therefore, this means less storage usage and more efficient root cause analysis.

5.2.6 Spectrum-Based Fault Localization

Spectrum-Based Fault Localization (SBFL) is a technique used to identify faulty components in a system by analyzing execution traces from both successful and failed runs. It calculates a suspiciousness score for each component (e.g., function, service, or span) based on how often it appears in failing versus passing executions. Components that appear more frequently in failed traces are considered more likely to be the root cause of the issue [85, 136–139]. When a faulty program P gets tested with some test set that includes at least one failure test, SBFL first records which parts of code are covered by each test case ($O \in P$). Once this is done, it calculates four important numbers for each component: O_{ef} (failed tests using O), O_{ep} (passed tests using O), O_{nf} (failed tests skipping O), and O_{np} (passed tests skipping O). Then these numbers feed into a suspiciousness score calculation (e.g., using Tarantula), which ranks for how likely it is that each component is faulted. The same concept applies to tracing issues in microservices: requests are routed through multiple services, and the one linking more failed requests (and fewer successful) is likely to be the root cause.

SBFL is widely used for automated debugging and root cause analysis in both software systems and distributed environments [85, 136–139]. Some previous approaches [85, 140] apply SBFL to distributed systems by treating each request as a test case and each span as a component. They assign a suspiciousness score to each span to find the ones most related to failures. There are several formulas used for this, including Tarantula, Ochiai, Overlap,

Wong2, and Goodman. Inspired by the work in [85], we use Tarantula, which shows the highest accuracy compared to the others for our approach.

The basic SBFL method has some limitations. It counts how many failed and successful requests include each span but does not consider different request types [109]. When we have different request types, the accuracy of SBFL is reduced. Furthermore, test cases are designed by testers and are balanced, but all the requests are not balanced; this means that some requests appear more often compared to other types of requests Some request types appear more often than others, even when they call the same spans. This creates bias in the suspiciousness scores. Since all traces have equal weight, using SBFL directly for root cause analysis in microservices may cause poor results.

In order to solve these problems, the article [109] applies the PageRank algorithm to assign anomalous and normal scores for each span. The evaluation traces from failed and successful requests are then performed by the algorithm in parallel, ranking the spans according to their contribution to the actual failure. This step increases the accuracy of the root cause detection.

5.2.7 Page Rank Algorithm

PageRank is a method to measure the importance of nodes in a graph. It represents a random walk originating from the source node s and traversing connected nodes through edges. At each instant, the random walker may either follow the edge or jump back to s with a fixed probability. Accordingly, Personalized PageRank focuses on nodes that are important or closely connected to a specific source node s. In distributed traces, the nodes may represent spans or services, and the score for node d indicates how closely related it is to the source s, such as an anomalous span. [85, 140].

To address the limitations of spectrum-based analysis—specifically, that it treats all traces equally and is affected by some traces showing up more often than others, which can lead to biased results [85, 117],—Personalized PageRank (PPR) is introduced as a span scoring method. In this context, the trace data is modeled as a graph, where nodes represent services (spans) and edges represent communication or call relationships between them. PageRank [141] is an algorithm that ranks nodes in a graph by simulating a random walk, where the importance of a node increases if it is linked by other important nodes. PPR enhances the standard PageRank by applying a preference vector, which guides the random walk process toward specific nodes of interest-typically those more likely related to abnormal traces. This approach allows the algorithm to assign higher scores to spans that appear more relevant for fault localization, especially those in abnormal traces while reducing the impact of spans that

appear frequently but contribute little to identifying the root cause of latency issues. PPR improves on this by being able to identify the important operations as well under normal and abnormal traces.

5.3 Methodology

The overview of our architecture is illustrated in Figure 5.1. It consists of 4 modules: a trace collector, a request structuring module, an anomaly detector, and a root cause localizer. Each module is explained in detail in the following subsections.

5.3.1 Trace Collector

In distributed systems, high-level tracing tools such as OpenTelemetry, OpenTracing, and OpenCensus collect request data such as trace IDs and latency. Some studies, as described in Section 5.1, find performance issues in the Service Level Objective (SLO), but this approach lacks the granularity needed for root cause analysis. Low-level tracing is required to capture system metrics like CPU and memory through monitoring tools such as LTTng and Perf.

Some existing approaches use monitoring tools such as Prometheus [4], Datadog [5], and etc. that cannot link system metrics to specific spans. They do not show which kernel events occur during span execution. Previous work [3, 83] solves this by using double instrumentation to collect kernel events tied to process IDs for each span, and a visualization tool [6] that can sync the clocks between different machines, but they need careful consideration for storage space and selecting a subset of kernel metrics.

To address this, we adopt a hybrid approach. Our Trace Collector module collects distributed traces: $T = \{T_1, T_2, \ldots, T_n\}$, where each trace T_i consists of a sequence of span calls: $T_i = \{S_1^i, S_2^i, \ldots, S_m^i\}$, with S_j^i representing a span in trace T_i. Each trace has an end-to-end latency: $L = \{L_1, L_2, \ldots, L_n\}$.

The Similar Structure Extractor module groups the traces based on execution patterns and detects latency variations. If a trace T_i shows unusual latency L_i, the anomaly detector module triggers an event to identify the affected span S_j^i. Trace Collector then extracts system metrics related to the suspicious services, like a set of system metrics:

$$M = \{M_1, M_2, \ldots, M_n\}$$

4. `https://prometheus.io/`
5. `https://www.datadoghq.com/`
6. urlhttps://eclipse.dev/tracecompass/

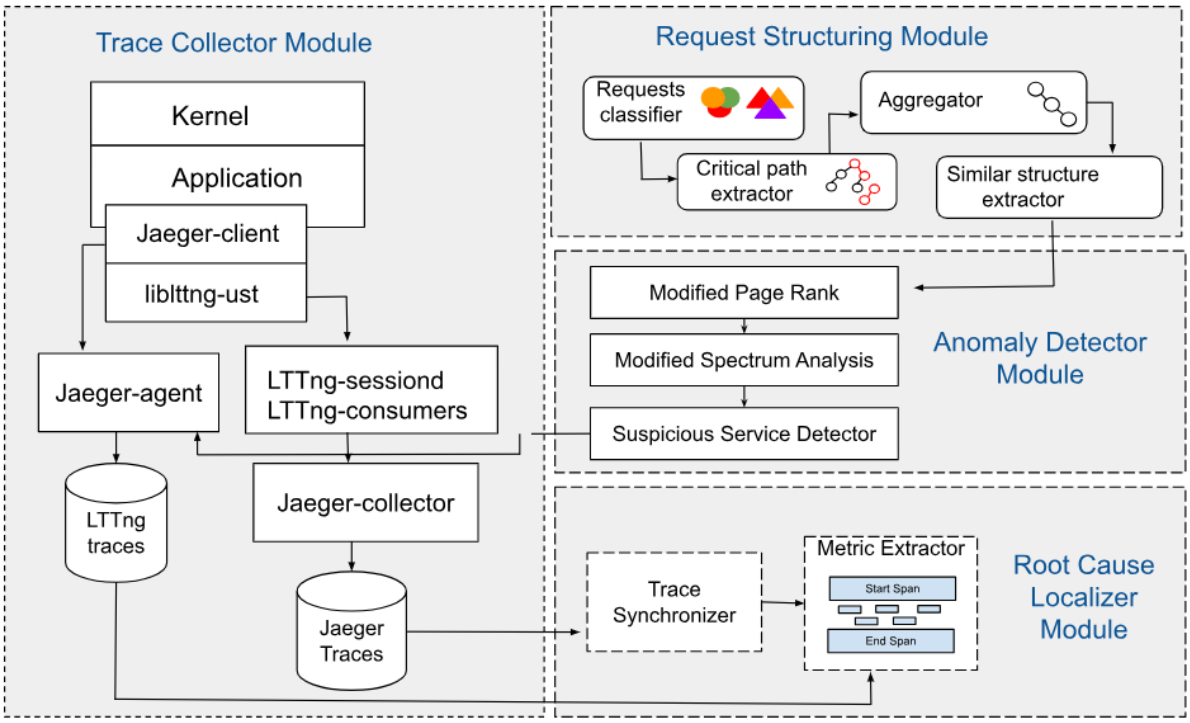


Figure 5.1 Software architecture consists of 4 modules: a trace collector, a request structuring module, an anomaly detector, and a root cause localizer. The details are explained in Section 5.3.

where each M_j represents system metrics such as CPU usage, memory consumption, and disk activity for a specific span S_j^i:

$$T_i = \{S^i_{\text{start}}, M_1, M_2, \ldots, M_k, S^i_{\text{end}}\}$$

The Trace Collector only collects system metrics for services that have been affected, to avoid unnecessary overhead on services that have not been affected. This approach minimizes the space storage for system metrics.

5.4 Request Structuring Module

This module consists of four sub-modules. Each module is explained in detail in the following subsections.

Request Classifier

In order to categorize execution times into either normal or abnormal, we basically classify requests by type, for example login requests and signup requests. The root span, generally

an API endpoint, identifies the type but users may also define other custom rules. The advantage of this step is that we can group and compare similar request types and calculate expected latency based on their type. In contrast, traditional approaches mostly calculate the expected latency across all requests [142]. These approaches detect general trends but often miss small anomalies in specific request types. Their models cannot adapt to the variations between different requests.

For instance, a simple `GET /status` request with an average response time of 50 ms and a complicated `POST /checkout` request averaging 800 ms could both affect the same mean. If the `GET /status` request starts experiencing higher latency at 200 ms, it would still be lower than the global mean and be left unnoticed. Such factors lead to real issues being overlooked.

Critical Path Analysis in Distributed Systems

Our goal is to identify the spans that have a significant impact on the overall request latency and, as a result, affect user experience. To do so, we aim to group similar requests and, within each group, define a normal range of execution.

Any other requests might then be categorized as anomalous requests. Some earlier studies use Directed Acyclic Graphs (DAGs) to model dependencies between spans or trace how errors propagate across services. They compute the expected latency by summing the execution time of each span within the DAG, for instance, which defines the expected latency L_{expected} of a trace as:

$$L_{\text{expected}} = \sum count_o \cdot (\mu_o + n \cdot \sigma_o)$$

Where $count_o$ is the number of times operation o appears in the trace. μ_o is the average latency associated with that operation, σ_o is the standard deviation, and n is the adjustment of the upper limit. However, DAGs do not sufficiently represent actual execution behavior; the dependencies are expressed without indicating how the spans may run in parallel. Furthermore, DAGs do not allow any comparison of execution patterns for similar requests. All requests in a distributed environment have a parent-child relationship that runs parallel to the child spans. The total end-to-end latency is not equal to the sum of all spans but is actually defined on the critical path—the longest sequence of dependent spans, which controls when the request actually finishes.The Trace Collector only collects system metrics for services that have been affected, to avoid unnecessary overhead on services that have not been affected. This approach minimizes the space storage for system metrics.

The important aspect of the critical path is that it regards only the spans that affect the

overall latency of the request directly. For example, a request in which the parent is calling Service A and Service B in parallel. Service A is taking 180ms, while Service B is taking 100ms. Service A will form part of the critical path. However, should the Service B become slow compared to A, the shift in the critical path includes Service B. In both these cases, only the slowest span is delaying the parent and contributing to end-to-end latency.

By looking only at the critical path, we do not need to analyze all spans in a trace. We focus on the ones that limit how fast the request can finish. It lowers the computational cost by reducing the number of spans that do not affect the total latency. We can group similar requests and calculate the normal range of execution for each path. This method helps detect anomalies and optimize performance more effectively [135, 143, 144].

To formulate our approach: A parent service S_p invokes multiple child services that could be executed in parallel. The self-time of S_p, denoted as $\tau_{\text{self}}(S_p)$, is the time spent only on performing its own work without the overlapping time of child executions.

In case, the child service $S_1, S_2, ..., S_m$ run at the same time. Then initial step is to check for overlapping time executions. By definition two child services are within execution periods at same time, $S_i \parallel S_j$: There is overlap in their execution times if the execution times follow:

$$T_{\text{start}}(S_i) < T_{\text{end}}(S_j) \quad \text{and} \quad T_{\text{start}}(S_j) < T_{\text{end}}(S_i).$$

We calculate the total overlapping execution window as:

$$O(C) = \sum_{\text{merged intervals}} (\text{end} - \text{start}).$$

The corrected self-time of the parent is:

$$\tau_{\text{self}}(S_p) = \tau_{S_p} - O(C).$$

This makes sure that the parent does not count time covered by overlapping child services. Figure 5.3 shows this process. The total execution time of S_p is:

$$L(S_p) = \tau_{\text{self}}(S_p) + \max_{P_j \in \text{Parallel}} (\tau_{P_j}) + \sum_{C_k \in \text{Seq}} \tau_{C_k}.$$

P_j represents parallel child services, and C_k represents sequentially dependent services. The end-to-end latency for a request passing through S_p is:

$$\lambda(S_p) = T_{\text{end}}^{\text{critical}} - T_{\text{start}}(S_p).$$

$T_{\text{end}}^{\text{critical}}$ is the completion time of the last service in the dependency chain.

Aggregator

In this module, considering that in the critical path extraction module we eliminated branches that were executed in parallel within the graph, the remaining span calls form a sequence similar to a call stack. This means that each span calls its subsequent span sequentially, with no parallelism. As a result, if a performance degradation occurs at any point in the chain, all upstream spans will be impacted due to their dependency on the downstream spans. To identify requests that exhibit similar behavior in a distributed system, we focus on aggregating identical spans that execute immediately one after another and share exact names. For example in Figure 5.4, a user login request that requires retrieving data from the cache and, in case of an error, attempts this retrieval up to three times. Such requests—whether they encounter an error once, twice, or even three times in the cache—should be grouped together. However, requests that, after three cache errors, call another source, such as a database should be placed in a separate group. Therefore, for each critical path, we aggregate spans that are executed immediately one after another, if they share the same name.

Similar Structure Extractor

In this module, to define the expected latency for abnormal requests, we group critical paths that follow the same sequence of span calls. For example, when a user logs in for the first time, the system may fetch user data from the database. In a similar login request, the system may fetch the same data from the cache. If we group both critical paths together to define the normal execution range, the result may be inaccurate. This happens because database access usually takes more time than cache access. However, both requests follow normal behavior for their own path. Mixing them in one group can cause false anomaly detection or missed issues. As shown in Table 5.1, there are different strategies for grouping sequences of names based on their similarity [145]. We use exact matching because it is fast and efficient for this task. This method uses hash-based lookup, which works in O(N) time complexity.

Table 5.1 Comparison of Sequence Grouping Strategies

Method	**Best For**	**Complexity**
Exact Matching (Dictionary)	Identical sequences	$O(N)$
Jaccard Similarity	Partial overlap	$O(N^2)$
Levenshtein Distance	Small variations (insert/delete/swap)	$O(N^2)$
Clustering (Jaccard Distance + ML)	Large datasets with unknown groups	$O(N \log N)$

5.4.1 Anomaly Detector

In this module, we focus only on requests with a successful status. Some requests have shorter latency than expected because of exceptions or errors that stop them from reaching all required services. Other requests take longer than usual because they wait for a third-party service that does not respond, causing high tail latency and an error status. In this step, we analyze only requests with a successful response code. The goal is to find the expected latency for each group extracted from the Section 5.4 and finding the suspicious span in terms of latency within each group of critical paths.

This module sets a maximum threshold for the expected latency for each group. When the latency of a sequence of call spans exceeds this threshold, it is considered an abnormal request. Traditional anomaly detection approaches focus on the average latency in Service Level Objectives (SLO) [41, 79, 146, 147]. However, these traces consist of a diverse range of operations, each with different processing times [85]. This may cause inaccuracy; some operations may naturally take longer than others (e.g, database operations typically take longer than a cache), and both are normal.

Inspired by the work [85,146], we extend the expected latency estimation formula to explicitly handle parallel execution, and self-time extraction for ther requests inside of each group.

$$L_{\max} = \sum (\text{count}_o \cdot (\mu_{\text{self},o} + n \cdot \sigma_{\text{self},o})) \tag{5.1}$$

Where $L_{\max}$ is the maximum latency, summing over all operations (spans) o in the critical path. count_o shows the number of times an operation o appears in the critical path. The mean execution time $\mu_{\text{self},o}$ and the standard deviation $\sigma_{\text{self},o}$ are calculated based on all occurrences of the operation across all requests of the same group, not just those within the critical path. The parameter n is a tuning factor, which we set to 0.03 based on our evaluation.

After identifying requests with execution times exceeding the expected latency, the next step

is to pinpoint the suspicious spans that contribute most to the delay. which is explained in the following section.

Personalized PageRank for Anomaly Localization

To assign an importance score to each span—highlighting those that are more likely responsible, as discussed in Section 5.2.6—we apply the PageRank algorithm to our critical path graph. Unlike the traditional PageRank approach [85], which is designed for homogeneous graphs, that approach also does not consider parallelism in distributed systems, which affects request execution time. We adapt the algorithm to our context by aggregating identical spans at each level of the critical path.

Graph Model for Root Cause Localization In our approach, the distributed request flow is represented as a directed graph $G = (V, E)$, where:

— V represents services in the system.

— E represents dependencies between services along the critical path.

— Vertex weights show the aggregated execution time of that service.

The transition probability between two services s and t is:

$$A_{st} = \begin{cases} \frac{1}{|O(s)|}, & \text{if } t \in O(s) \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

where $O(s)$ is the set of outgoing dependencies from service s. This matrix describes how requests move through the system and helps measure service influence on execution delays.

We define an anomaly vector u to highlight affected traces. The Personalized PageRank (PPR) score is computed as:

$$v = (1 - d)Av + d \cdot u \tag{5.3}$$

where d is the damping factor, which controls the balance between dependency influence and random jumps. The transition matrix for service aggregation:

$$A = \begin{bmatrix} \omega A_{\text{services}} & A_{\text{requests}} \\ A^T_{\text{requests}} & 0 \end{bmatrix} \tag{5.4}$$

where:

— A_{services} represents service-level dependencies within critical path.

— A_{requests} links service executions to anomalous requests.
— ω ($0 \leq \omega \leq 1$) controls the weight of service interactions.

After computing PageRank scores, the method identifies the most probable root cause spans. If an anomaly affects multiple spans, the highest-ranked node in the graph is prioritized.

Spectrum-Based Fault Localization

To complement the structural insights provided by Personalized PageRank, we incorporate *Spectrum-Based Fault Localization (SBFL)* to statistically estimate how likely each span contributes to latency anomalies. SBFL does this by comparing how frequently each span appears in *abnormal* versus *normal* request traces.

Each span o is evaluated using a spectrum tuple consisting of four components:

— O_{ef}: weighted number of **abnormal** requests that include span o
— O_{ep}: weighted number of **normal** requests that include span o
— O_{nf}: weighted number of **abnormal** requests that **exclude** span o
— O_{np}: weighted number of **normal** requests that **exclude** span o

Unlike traditional SBFL, which treats all traces equally, we assign each request a weight based on its importance using the PageRank scores computed in Section 5.4.1. This allows us to prioritize more informative traces and reduce bias caused by frequent or noisy patterns.

The weighted spectrum values for each span o are calculated as follows:

$$\begin{aligned} O_{ef}(o) &= F(o) \cdot N_{ef}(o) \\ O_{nf}(o) &= F(o) \cdot (N_f - N_{ef}(o)) \\ O_{ep}(o) &= P(o) \cdot N_{ep}(o) \\ O_{np}(o) &= P(o) \cdot (N_p - N_{ep}(o)) \end{aligned} \tag{5.5}$$

Here, $N_{ef}(o)$ and $N_{ep}(o)$ represent the number of abnormal and normal requests, respectively, that include span o. The values N_f and N_p are the total numbers of abnormal and normal requests. The functions $F(o)$ and $P(o)$ denote the PageRank scores of span o derived from the abnormal and normal trace graphs.

To compute how suspicious each span is, we apply the *Tarantula* formula:

$$\text{Suspiciousness}(o) = \frac{\frac{O_{ef}}{O_{ef}+O_{nf}}}{\frac{O_{ef}}{O_{ef}+O_{nf}} + \frac{O_{ep}}{O_{ep}+O_{np}}} \tag{5.6}$$

A high suspiciousness score indicates that the span is frequently involved in slow or abnormal

requests that rarely appear in normal ones—making it a likely root cause.

5.4.2 Root Cause Localizer

The next step, after identifying the suspicious spans that play a key role in increasing the overall request latency, is to understand what really caused the problem. This is done by incorporating system metrics with high-level service information. Inspired by previous work [3,83], we apply a hybrid approach to associate system metrics with the suspicious spans and solve the issue of synchronization. To overcome space limitations, instead of collecting system metrics for the entire request, we focus only on a specific suspicious span. We optimized the process by minimizing storage requirements and reducing the computational cost for evaluating system metrics from the kernel events.

For the technical part of our work, we use a feature in LTTng called event rules. We configure LTTng to monitor user-space events based on the name of the suspicious span. When this span is triggered, LTTng automatically starts to collect the kernel-level events corresponding to the process and thread ID of the targeted span. In the operating system, there are two layers: user space and kernel space. High-level traces and logs are generated in user space, and system metrics are collected in the kernel. The kernel events captured by LTTng are then programmatically converted to meaningful system information such as CPU usage, memory usage, etc.

Assume S_{sus} represents the set of suspicious services identified through anomaly detection. When a service $S_i \in S_{\text{sus}}$ is detected, we define the triggering condition as:

$$\mathcal{T}(S_i) = \begin{cases} 1, & \text{if } S_i \text{ is flagged as suspicious,} \\ 0, & \text{otherwise.} \end{cases}$$

Once $\mathcal{T}(S_i) = 1$, LTTng tracing is activated for S_i. At this point, we collect two complementary modalities of data: high-level trace events and low-level system metrics $M(S_i)$, forming a combined dataset:

$$D(S_i) = \{\text{Traces}(S_i), M(S_i)\}$$

Here, $\text{Traces}(S_i)$ includes user-space events such as span start and end times, while $M(S_i)$ includes fine-grained kernel-level resource metrics such as CPU usage, and I/O wait time.

An example sequence of these events may look like:

$$S1_{\text{start}},\ \text{system metrics}, S1_{\text{end}} S2_{\text{start}},\ \text{system metrics}, S2_{\text{end}}$$

By combining system metrics with high-level trace events, we create a mapping $\mathcal{R}$ that links performance deviations to the related service execution:

$$\mathcal{R} : M(S_i) \times \text{Traces}(S_i) \rightarrow C(S_i)$$

where $C(S_i)$ is the set of potential root causes for performance degradation in service S_i. This integration helps us find the exact source of the problem. It also lets us compare the resource usage of the suspicious span when it has high latency with when it runs as expected, to see the differences. This clear view makes root cause analysis more accurate in distributed systems. Figure 5.2 shows an overview of the suspicious span identified by our tool.

Span Metrics Summary ×

Start Time	End Time	Duration	SpanId	OperationName	TraceID	ThreadID	WAITBLOCKED	RUN	RUNSYSTEMCALL	INTERRUPTED	WAITCPU
13:05:09.198 646 430	13:05:09.374 512 583	175,866,1	91465661043128202	/driver.DriverService/FindNearest	90201644011432979 80	383508	174119015	95684	475903	23221	333116
13:05:09.198 896 400	13:05:09.374 290 636	175,394,2	60092067649727585 79	/driver.DriverService/FindNearest	90201644011432979 80	383516	174277578	38710	238363	0	563938
13:05:15.361 879 445	13:05:15.579 275 068	217,395,6	71713935518926143 63	/driver.DriverService/FindNearest	22200193810459615 49	383516	216237757	52248	332704	17901	311096
13:05:15.362 098 307	13:05:15.578 981 546	216,883,2	21554697847165519 88	/driver.DriverService/FindNearest	22200193810459615 49	383622	215306275	69406	342544	4697	743721
13:05:42.253 617 267	13:05:42.479 626 922	226,009,6	23126858837229399 25	/driver.DriverService/FindNearest	51088597797878797 53	385061	225290997	51069	207653	55118	53877
13:05:42.254 911 191	13:05:42.479 350 172	224,438,9	67571935254146149 64	/driver.DriverService/FindNearest	51088597797878797 53	385063	223559509	51885	267419	0	300865
13:05:48.628 040 351	13:05:48.814 009 957	185,969,6	21967276471621138 78	/driver.DriverService/FindNearest	57755653101692493 97	385060	5448481876	19813	261598	9626	30044
13:05:48.628 267 468	13:05:48.813 793 200	185,525,7	25322004925745708 10	/driver.DriverService/FindNearest	57755653101692493 97	385051	184362923	74948	264954	0	227404

Figure 5.2 An overview of suspicious spans identified by our tool with their corresponding Trace ID, Span ID, and Duration. The table summarizes major performance figures relative to key system metrics, such as WAITBLOCKED, RUN, RUNSYSTEMCALL, INTERRUPTED, and WAITCPU time, which assist in diagnosing potential performance bottlenecks.

5.5 Experimental Evaluation

In this section, we demonstrate how our approach reduces the number of spans needed for evaluation, thereby lowering storage space requirements and computational costs, while still being sufficient for root cause analysis and localization. We also show that our approach is effective in identifying real-world performance issues.

5.5.1 Microservice Applications

TiDB[7] is an open-source distributed SQL database created at PingCAP. The architecture consists of several services in the TiDB organization: TiDB (SQL layer), TiKV (storage),

7. `https://github.com/pingcap/tidb`

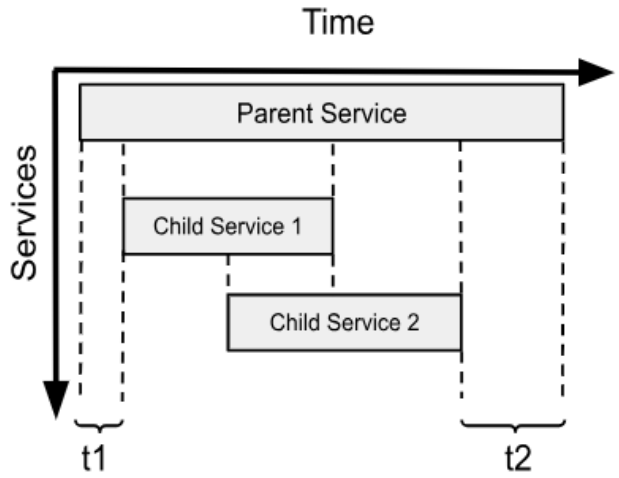


Figure 5.3 The parent-child relationship and the parallel service execution is depicted in this diagram. The Parent Service has a self-time of $t_1 + t_2$, representing time spent without active child services.

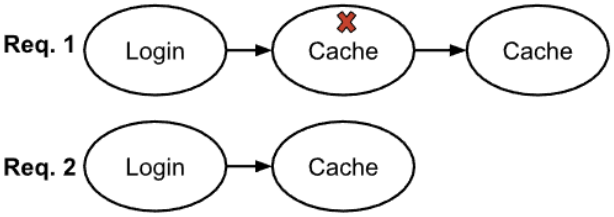


Figure 5.4 Two requests of the same type but with different structures. By aggregating identical service calls that occur consecutively, the two traces can be grouped and treated as the same.

PD (Placement Driver), TiFlash (analytics), and TiCDC (data capture). The system is instrumented with OpenTracing for monitoring query performance and tracing request flows.

HotROD (Rides on Demand) [8] is another open-source application created by Uber as part of the Jaeger tracing project. It comprises seven microservices: frontend, customer, driver, route, ride, location, and Redis. The entire application is fully instrumented with OpenTracing. We chose these software systems because they are open-source, instrumented with OpenTracing, and have been used in previous research [3, 83].

We have instrumented the OpenTracing source code with LTTng in order to gather system-level metrics. This setup allows us to capture all the necessary metrics, including process IDs and thread IDs, thus allowing us to correlate between application-level traces and system

8. `https://github.com/jaegertracing/jaeger/blob/main/examples/hotrod/README.md`

behavior. We also removed the intentionally injected sleep code from the HotROD service to properly evaluate its impact on performance degradation.

5.5.2 Datasets

To evaluate our work, we adopted two complementary approaches. First, we leveraged a publicly available dataset from prior research [118], which contains traces from two open-source microservice applications: TrainTicket, a system for checking, booking, and purchasing railway tickets, it contains 41 microservices. OnlineBoutique, an e-commerce platform. This dataset includes 11 microservices and a total of 56 injected faults—42 related to resource issues and 14 from code defects.

5.5.3 Experimental Platform

All the experiments were conducted on two computers consisting of AMD Ryzen 7 5700G, with a Radeon Graphics CPU operated at 1400 MHz, and 32 GB of DDR4 memory. The OS installed is Ubuntu 20.04.1 based on Linux kernel version 5.15.0-71, while LTTng is version 2.13.9-1.

5.5.4 Evaluation metric

To evaluate the performance of our model, we employed several standard metrics. Top@K evaluation, K refers to the number of highest-ranked items considered from the output list generated by the model. We also used precision, recall, and F1 metrics to measure the effectiveness of our approach.

5.5.5 Fault Injection

We used the existing datasets from the previous work as described in Section 5.5.2. In those datasets to simulate performance degradation within the software, a technique called fault injection was used. In this process, faults were purposely introduced into the program to analyze how it would cope with performance degradation. The process followed the methods used in earlier studies [45,45,85,148]. Specifically, CPU contention was applied to open-source applications using the same approach as in previous research [45, 45, 85, 118, 148]. We also simulate a performance issue in a real-world application to evaluate whether our approach can accurately identify the problem [3].

5.5.6 Impact of Trace Number and Size

In this section, we have shown how our solution can outperform other approaches that require full trace evaluation at both ends of the service layer and the metric collection layer, while nevertheless requiring less data for processing and root cause analysis. Chart 5.5 shows that analyzing just the critical path alone could reduce the number of examined spans up to 22.6%. Table 5.2 shows that selectively capturing system metrics associated with the suspicious node significantly reduces the required storage volume by 99%. This optimization relies on measuring only those metrics relevant to the affected service, rather than collecting all metrics. The reduction in storage requirements by 99% also shows the effectiveness of this method, which is of great importance for large-scale systems, where scalability and resource limitation are important considerations.

Table 5.2 Summary of trace size, number of requests, spans, and the contributions of user-space trace (UST) and kernel events to the total trace size. The results show a significant reduction in kernel trace data, decreasing from hundreds of megabytes to a few hundred kilobytes—over 99% reduction—leading to lower storage and analysis overhead.

Trace Size	**Req.**	**Spans**	**UST**	**Kernel**	**Reduced Kernel**	**Kernel Size Reduction**
29.6 MB	50	250	428 KB	29.2 MB	75 KB	99.74%
89.3 MB	250	1,250	1.3 MB	88.0 MB	375 KB	99.57%
326.4 MB	500	2,500	3.0 MB	323.3 MB	225 KB	99.93%
978.6 MB	3,000	150,000	17.5 MB	961 MB	300 KB	99.97%
1.7 GB	5,000	250,000	29.2 MB	1.7 GB	375 KB	99.98%

5.5.7 Anomaly Detector Effectiveness

Figure 5.3 shows the results of our method on three datasets: HotRod, Train Ticket, and Online Boutique. Each dataset is explored for one root cause and two root causes. The recall value is always on the higher side, with 100% for HotRod and above 90% in the others; precision is also high, with 97% for one root cause in HotRod and 85% for two root causes in Online Boutique; the F1-Score stays between 88% and 98%; the results demonstrate the robustness of our method across various scenarios in determining root causes, even in the presence of more than one issue.

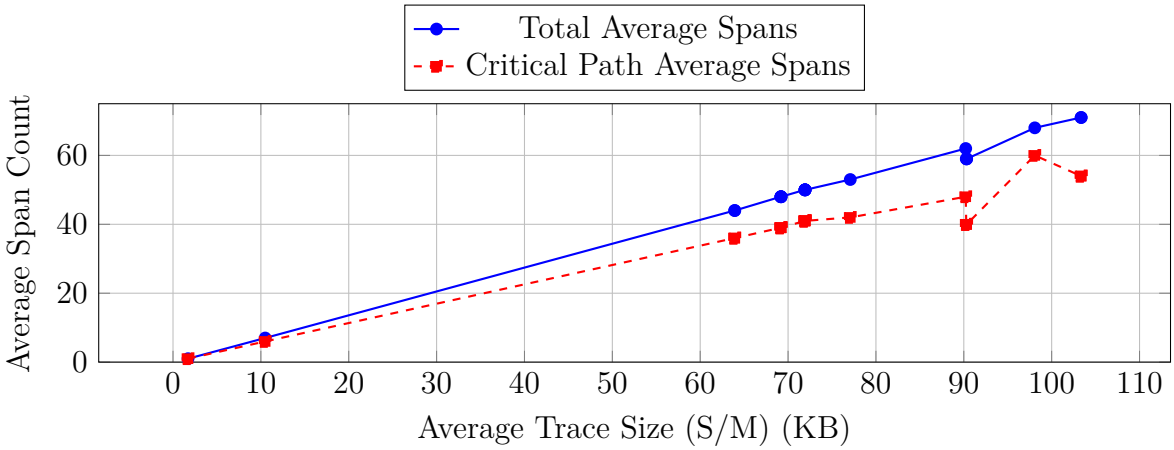


Figure 5.5 This chart shows the average number of spans per trace across traces of varying sizes. It demonstrates that our approach reduces the number of spans used for evaluation by up to 22.6%, which can lead to lower computational cost and reduced disk space usage.

Table 5.3 The table shows recall, precision, and F1-score for our method on HotRod, Train Ticket, and Online Boutique. We test with one root cause and two root causes. The results show high scores in all cases, even when there are two faults.

Test Case	**Recall (%)**	**Precision (%)**	**F1-score (%)**
One RC HotRod	100	97	98
Two RC HotRod	100	94	97
One RC Train Ticket	97	91	94
Two RC Train Ticket	96	93	94
One RC Online Boutique	92	89	90
Two RC Online Boutique	90	85	87

5.5.8 Root Cause Localizer Effectiveness

We evaluate the effectiveness of our approach in diagnosing real-world open-source software performance issues, inspired by a reported problem during an update of TiDB [9]. Specifically, we evaluate two key capabilities: Fault localization—whether the tool can accurately identify the problematic service. Root cause analysis—whether it can correctly determine the nature of the issue. To establish ground truth, we relied on a use case from previous research [3], where a TiDB update caused the execution time of index requests to increase significantly, from 4,397 ms to 7,663 ms. We reproduced this scenario by applying a synthetic workload using SysBench (32 threads, 1-minute duration). Our tool successfully detected the degraded service, confirming its ability to pinpoint performance anomalies in Figure 5.6.

9. TiDB issue #44584

Span Metrics Summary ×

Start Time	End Time	Duration	SpanId	OperationNam	TraceID	ThreadID	WAITBLOCKED	RUN	RUNSYSTEMCALL	INTERRUPTED	WAITCPU
14:56:22.014 030 756	14:56:22.014 268 883	238,127	8890769587297649618	SQL SELECT	81960569281468135O5	535577	1625923	162623	32207	0	0
14:56:23.237 082 731	14:56:23.237 210 791	128,060	2241661113121453339	SQL SELECT	82402840912154053	535605	28342	95853	40528	0	13564
14:56:24.381 699 096	14:56:24.381 889 914	190,818	11487966662841385З0	SQL SELECT	84325955367391488I7	535584	155892	126371	52110	0	0
14:56:25.365 444 359	14:56:25.365 583 691	139,332	6537906447525474413	SQL SELECT	67579850552784207O5	535576	27742	130249	29729	0	11731
14:56:26.246 263 126	14:56:26.246 376 048	112,922	68397995832141871Б1	SQL SELECT	3501501031322880325	535581	47869	84315	23320	0	0
14:56:27.261 413 182	14:56:27.261 591 708	178,526	8444477251505301839	SQL SELECT	4165744294368763466	535610	176941	81798	38957	0	0
14:56:28.142 158 392	14:56:28.142 299 668	141,276	3459963605436634325	SQL SELECT	58467641343801280Б4	535584	177052	53332	22766	0	0

Figure 5.6 The table shows system metrics for the suspicious SQL SELECT span. Based on a real issue from prior research [3], we simulated the scenario, collected related kernel events, and extracted span-specific metrics. Results confirm our method can detect performance issues with fewer metrics and accurately identify root causes.

5.5.9 Comparison

We test our approach against three state-of-the-art methods: MicroRank [85], Nezha [118], and HemiRCA [133]. For both Nezha and HemiRCA, we use the service-level results. To enable a fair comparison with Nezha, we adapt our method to return pod names instead of service names, aligning with their granularity level. Table 5.4 shows that our method achieves slightly better or comparable performance on both datasets. In terms of top-1 recall, our method shows a marginal improvement of 0.23% on the OnlineBoutique dataset and 0.03% on the TrainTicket dataset compared to the best-performing baseline (MicroRank). While the improvement in recall is not significant, it is noteworthy that our approach achieves comparable accuracy while reducing the data collection overhead and storage requirements. This demonstrates the efficiency of our method in maintaining high performance under resource constraints.

Table 5.4 Service-level recall (R@1, R@2, R@3) comparison across OnlineBoutique and TrainTicket datasets. Our method shows better or similar results compared to MicroRank, Nezha, and HemiRCA in both datasets.

Approach	**OnlineBoutique**			**TrainTicket**		
	R@1	**R@2**	**R@3**	**R@1**	**R@2**	**R@3**
MicroRank	74	74.15	79.58	61.98	62.68	62.65
Nezha	64.28	64.28	64.28	42.22	44.44	44.44
HemiRCA	71.3	73.4	78.4	60.49	61	61.78
Ours	**74.23**	**75.78**	79.44	**62.01**	**63.56**	**62.71**

5.6 Threats To Validity

One threat to validity is that our approach focuses on the performance degradation that affected the total latency of the request. If a service has a latency spike but is not on the

main path of the request, our method may not detect it. As a result, some performance issues may go undetected—particularly those that do not directly impact the end-user experience. While many studies in this field clearly define the range of abnormal requests they use, our method does not need this. We also notice that our approach works better when there are more normal requests than abnormal ones. Another point is that some related works did not share their source code, or the source code was not complete. Because of this, we re-implemented their methods based on our undrestanding of their papers, so our results may be slightly different from what they state in their papers. However, we made every effort to ensure that the comparison remained as fair and consistent as possible.

5.7 Conclusion

In this paper, we present a lightweight hybrid approach for root cause analysis in microservices. The method uses high-level traces and low-level system metrics together to detect performance issues. It focuses on the critical path of each request and ignores spans that do not affect end-to-end latency. The approach does not require the abnormal time window to be defined in advance. It also reduces data usage by collecting system metrics only for suspicious spans. Our method decreases the number of spans to analyze by 22.3% and lowers storage usage by more than 99%. We evaluate the method on three microservice systems: HotRod, TrainTicket, and OnlineBoutique. These results show that the method is effective and efficient for root cause localization in large-scale distributed systems.

Acknowledgment

We would like to thank AMD, Ciena, EfficiOS, Ericsson, Natural Sciences and Engineering Research Council of Canada, and Prompt for funding parts of this work. We also want to thank Arnaud Fiorini for helping us in this project.

CHAPITRE 6 ARTICLE 3: INSIGHTAI: ROOT CAUSE ANALYSIS IN LARGE HIERARCHICAL LOG FILES WITH PRIVATE DATA USING LARGE LANGUAGE MODELS

Authors Maryam Ekhlasi, Anurag Prakash, Michel Dagenais, Maxime Lamothe

Published at 4th International Conference on AI Engineering – Software Engineering for AI. (12 January 2025)

Abstract [Problem] As industries increasingly depend on complex software systems, efficient log analysis is essential for maintaining reliability and privacy. However, Identifying problems through logs is often time-consuming and costly for developers. [Background] Large language models (LLMs) can automate parts of log analysis, but challenges like limited computational resources and the frequent need to retrain LLMs due to the dynamic nature of software logs persist. External LLMs, such as GPTs, along with in-context learning techniques, can help reduce some of these issues, but other challenges, including token limitations, high token costs, and data privacy, remain. [Method] To tackle these challenges, we developed an automated pipeline that extracts log files and employs in-context learning, allowing the model to efficiently adapt to changes without extensive retraining. Our approach introduces a novel flame-graph-like method that reduces token usage, thereby lowering token-related costs and response latency while maintaining high accuracy. [Results] This solution allows industries to automate log analysis, minimize system downtime, and enhance performance, all while keeping data privacy and maintaining operational efficiency. [Conclusion] Our flame-graph-like methodology reduces input tokens by 93.61% and processing latency by 77.45%. Our anonymization results show an improvement of 138.63% over the baseline. This industrial experience report presents our approach to allow industries to balance token costs, maintain response accuracy, and ensure data privacy while relying on external LLMs without the need to manage computational resources directly.

6.1 Introduction

The complexity and size of modern software systems produce massive amounts of logs. logging is now crucial practice for maintaining software reliability [110–112]. They can reveal various issues during production, such as hardware failures, misconfigurations, poor coordination between services [3]. They can be also used for anomaly detection [47, 49, 113, 149], failure prediction [150, 151], and root cause analysis [152, 153]. Since these logs can grow to

gigabytes in size, manually reviewing them becomes impractical. Automating log analysis is therefore essential for efficiently identifying and addressing problems [154].

To address these challenges, Large Language Models (LLMs) have demonstrated considerable potential for automating log analysis processes within software systems [155]. However, despite their capabilities, recent LLMs, which can process large amounts of input data—referred to as the context window—are computationally expensive. This is because they require advanced GPUs with significant memory capacity to function efficiently [156–160].

On the other hand, language models struggle to effectively utilize information from long input contexts. Specifically, their performance tends to be best when important information appears at the beginning or end of the input, but drops significantly when the relevant information is located in the middle of long inputs [160]. Therefore choosing the best context window size is a challenge since it can be costly and directly affect the accuracy of the responses the LLM models produce.

Although, modern LLMs can automate the process of log analysis by processing and interpreting vast amounts of log data, their high computational costs and need for powerful hardware is another challenge for many companies. As a result, most companies choose to rely on external services, such as GPT models from OpenAI, to handle these tasks.

In addition to the challenges of token limits, context window, and computing costs, dealing with the privacy of log data are another big issues when using LLMs for log analysis. Logs often have sensitive details like IP addresses or user information that need to be handled carefully to follow privacy laws. Using LLMs, especially through third-party services, can risk exposing this private data [161–163]. Even when running LLMs on local machines, it is important to use data anonymization techniques to keep information safe. So, finding a balance between processing large amounts of logs efficiently and protecting sensitive data is a key challenge in using LLMs for log analysis. This shows the need for solutions that not only reduce token usage but also ensure privacy, which makes LLMs a secure and practical choice for organizations that manage large logs.

Based on our industrial experience, this experience report introduces a novel, adaptive approach to log file analysis using an LLM model. It focuses on solving four main challenges, listed in priority as token size limitation, data privacy with minimal loss of accuracy of LLM model responses, context window size tuning for higher accuracy, and prompt engineering for each log type and the chunks within each log. Here, *context window* size refers to the amount of text the model can consider at once, and tuning it means adjusting this limit to capture all necessary information for accurate responses. *Prompt engineering* involves designing specific questions or instructions for the model to help it better understand and respond to each type

of log.

The key contribution of this paper are as follows:

— In Section 6.3, we propose an adaptive approach that determines whether to analyze a specific log chunk, the entire file, or a defined time window. This decision-making process is designed to address user questions effectively for root cause analysis or suspicious event detection. This technique can be applied to any dynamic data, such as logs, to focus on relevant sections based on user queries. This approach reduces the token count per call, helping to lower costs.

— We introduce and provide our experience with a novel method inspired by the principles of flame graphs [164] to help reduce the issues of token limitation and cost associated with large-scale log analysis. The flame-graph-like approach reduces input tokens by an average of 85.53% across all logs, improving performance by 75% over the baseline pipeline latency. This technique enables users to save on token costs while maintaining accurate LLM responses.

— We introduce our experience with anonymizing log data to protect sensitive information, the accuracy of our method is 138.63% better compared to general mappings. Industries and researchers can apply a similar approach to address their own data privacy concerns, reducing hallucination and improving accuracy in LLM responses.

This research has led to the development of a chatbot solution that is already in being used by developers at Ciena. A general overview of the application is shown in Figure 6.1.

6.2 Background and Related Work

Large and complex software-intensive systems typically generate extensive log data during operation, primarily for troubleshooting and diagnosing issues [165]. However, analyzing such data at scale requires advanced techniques, as traditional methods are unable to handle the growing data volume and complexity effectively.

The recent advances in large language models (LLMs) have transformed natural language processing [56, 57]. LLMs generate text by predicting each word based on previous context using a probabilistic, autoregressive approach, making them effective for generating coherent and contextually relevant responses [58]. This capability, however, often requires adaptation techniques to handle specific tasks.

Several strategies exist for adapting language models, including fine-tuning and in-context learning. Among these, in-context learning (ICL) is a widely adopted and cost-effective approach. The core concept of ICL is learning by analogy, where a few example demonstrations

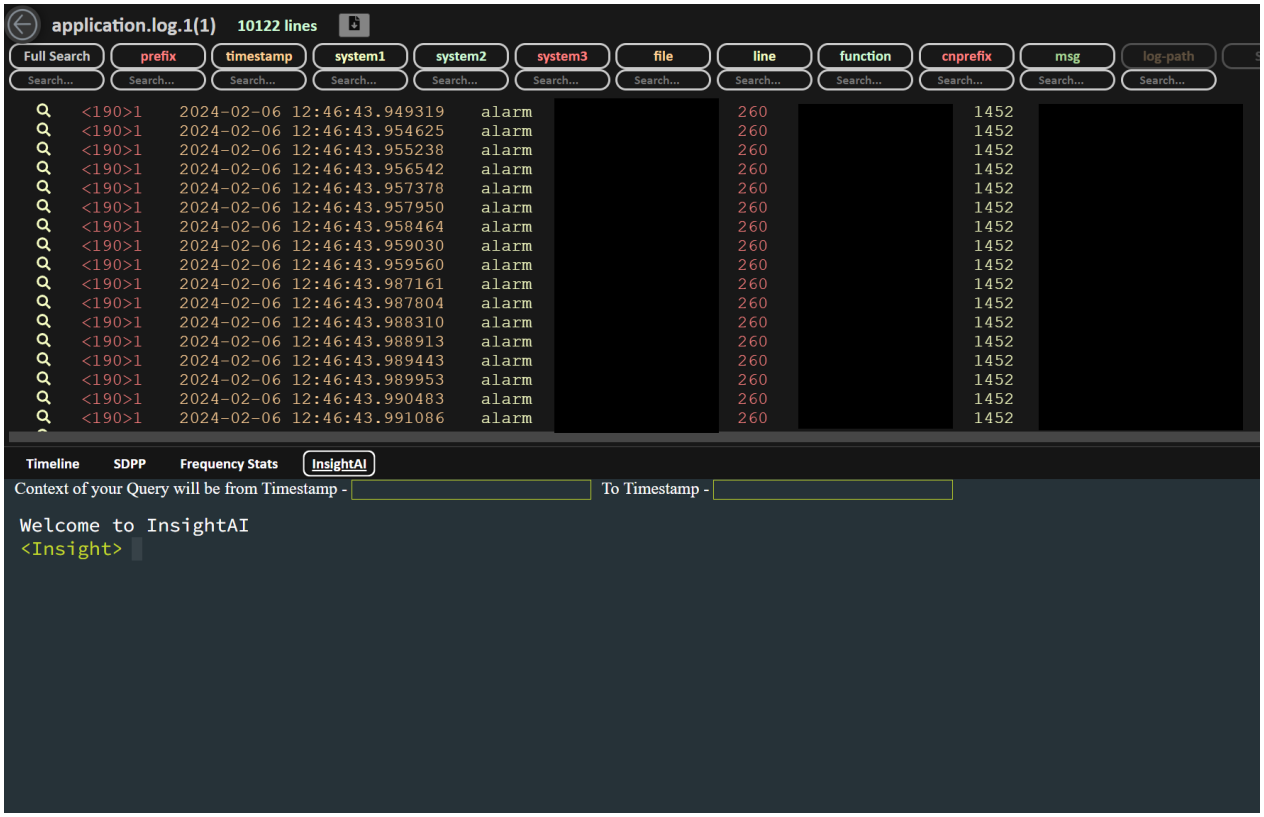


Figure 6.1 An overview of InsightAI: We developed a chatbot that allows users to ask any question related to the log, and the system processes the log to find the answer. Additionally, users can select a specific time range for evaluation. The logs are displayed with a filtering feature, which lets users hide unnecessary columns and focus on the desired data. For privacy concerns regarding our internal logs, we have intentionally hidden three columns in this image.

are used to create a contextual prompt. These examples are often formatted using natural language templates. ICL then combines the query question with this prompt context to form an input, which is passed to the language model to generate a prediction [59]. ICL can significantly lower the computational expenses of adapting models for new tasks, which enables language-model-as-a-service [60] and makes it practical for large-scale, real-world applications [59].

Building on ICL, few-shot learning further enables models to handle new tasks by showing only a few examples, while Chain of Thought (CoT) prompting improves understanding by breaking down answers into step-by-step logic. Zero-shot-CoT extends this by using a simple prompt like “Let’s think step by step”. It allows the model to handle complex tasks without any examples [61].

In this industrial experience report, we adopt zero-shot and few-shot CoT strategies in

prompts when using LLMs as a service, to help generate more accurate and detailed answers across a variety of logs.

Prior research categorizes large language models (LLMs) into three primary architectures: encoder-only, decoder-only, and encoder-decoder, each suited to specific tasks. Decoder-only models like the GPT series perform well in few-shot and zero-shot tasks due to their autoregressive nature, where each word is predicted based on previous words, making them ideal for text generation and question-answering [63, 64]. We chose GPT-4o for our pipeline due to its robust performance in handling complex context-dependent log-analysis tasks.

LLMs have many strengths but also some limitations. One key limitation is the context window length, which is the maximum amount of text the model can handle at one time [57, 63, 65]. This limit is set during training, and if it is exceeded, the model can become unstable. This is a big challenge for tasks that need a larger context. Adjusting the context window for each use case is important to keep performance high in large-scale log analysis.

To overcome the limitations of a context window, Retrieval-Augmented Generation (RAG) techniques improve model accuracy by integrating a retrieval system that finds relevant information from large datasets before generating responses. Previous work in this area, as highlighted in [166–170], shows the effectiveness of RAG. In our chatbot, implementing RAG enables the retrieval of relevant log chunks, which provide valuable context to the input of the model and contribute to improved accuracy.

To highlight the many uses of GPT-based chatbots, Qi et al. [171] did a survey and pointed out challenges like biases, hallucinations (creating information that sounds correct but is incorrect information), and privacy concerns. Macdonald et al. [172] also showed how LLMs can be useful in data analysis, especially when problems like context windows and privacy are handled. Since log data is very sensitive, keeping it private is still a big challenge.

To deal with privacy risks in log data, Aghili et al. [173] studied sensitive information in software logs. They found that attributes like IP addresses, timestamps, and port numbers are the most commonly considered sensitive. Their study also pointed out other important attributes, like configuration details and environmental data, which could cause privacy problems if shared. These results show the need for a good log anonymization method that not only meets privacy laws but also lowers the chance of exposing sensitive information. This is important for handling legal requirements and real-world problems faced by companies.

Below, we propose a novel industrial experience report that builds on previous work to address key challenges in using LLMs as a service for log analysis, including token limitations and the protection of sensitive information.

6.3 Proposed Solution

Large companies nowadays have complex software systems made up of multiple services that communicate with each other, generating an enormous number of log files. Each log file contains a vast amount of data, and there are important correlations between these logs that are key to identifying the root cause of issues. To effectively troubleshoot problems, experts first need to understand the architecture of the application, the hierarchy of the software, and how the logs are connected. Typically, they manually extract and analyze logs to find the problem, which is both time-consuming and costly. This process is particularly challenging for new hires who may not be familiar with the system.

To help with large log analysis in our industrial setting, we developed *InsightAI*, an intelligent chatbot powered by large language models (LLMs) and based on existing LLM technology. This approach aims to address the challenges developers and maintenance teams face in identifying root causes within large volumes of log data—especially in complex environments that require knowledge of software architecture and intricate application behaviors.

With InsightAI, users can query log files using natural language to find issues related to software execution, configuration settings, equipment status, and other unusual events.

To support this, we built a platform that gathers diverse logs and integrated ChatGPT-4o, an advanced LLM with a 128k token limit per API call, to assist developers in automatically detecting and analyzing log-based issues. We addressed two key challenges in this setup: (1) managing the token limit per API call and (2) protecting sensitive log data during processing. The result is a fully functional web application now in use at Ciena.

Figure 6.2 shows our pipeline architecture, which includes five modules. We describe each module in more detail below.

6.3.1 Decision Maker

The decision maker processes the user query and determines the appropriate strategy for limiting the number of tokens sent to the LLM. We have four strategies to address token limitations: explicit timestamp, manual time range selection, full content evaluation, and partial content evaluation, as explained below. In general, we use intent classification to determine the type of decision we need to make for choosing the best portion of the log file. Intent classification is the process of detecting and categorizing user intents in dialogue systems, including both known (seen) and new (unseen) intents, by using methods like capsule-based approaches and zero-shot learning to handle the user intents [174].

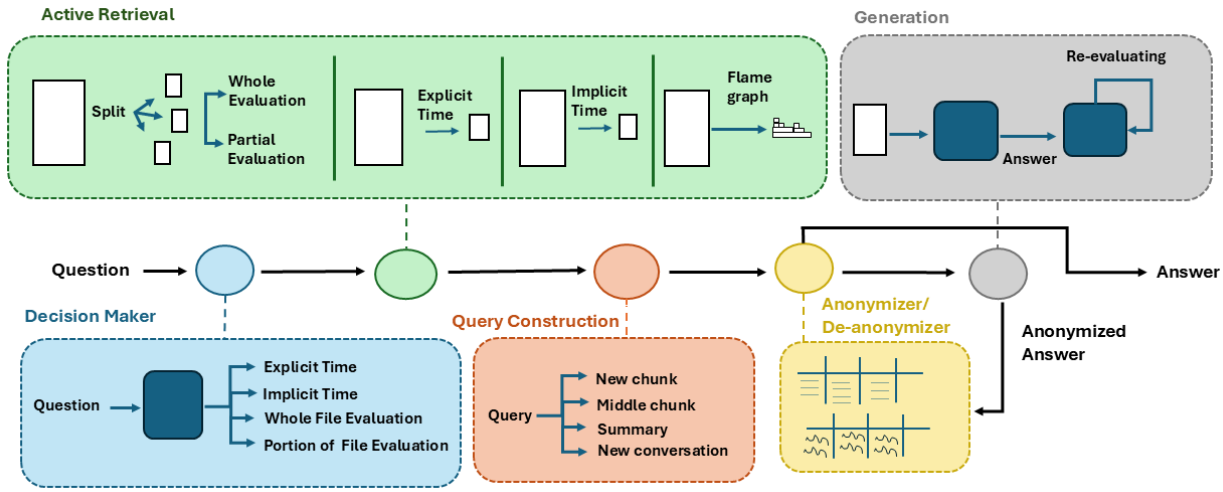


Figure 6.2 InsightAI Architecture with Five Modules: Decision Maker sets extraction scope based on user queries; Anonymization sanitizes sensitive information while retaining recognizability for the model; Active Retrieval extracts data chunks or generates flame graphs; Query Construction creates queries for summaries, new chunks, middle chunks, and new conversations; and Generation Module sends queries to the LLM and re-evaluates responses.

Explicit timestamp

If the query includes a timestamp, this module extracts the center time and passes it to the active retrieval system to apply a time windowing approach, centering on the user-provided timestamp. Based on expert input at Ciena, we use a default 30-second time window for analyzing logs, which can be adjusted to make the best use of tokens for each query. This way, we send only the relevant time frame to the model, reducing token usage. The time window can also be changed depending on the type of log or the specific issue being looked at.

Manual time range selection

Our empirical evaluations indicate that developers generally have a clear understanding of the approximate time range of error occurrences, which enables a more efficient and precise examination of relevant log segments. This module provides users with the capability to manually define the desired time window for targeted analysis. A general overview is illustrated in Figure 6.1.

Full or portion of content evaluation

To determine whether a query requires evaluation of the entire log file or just a portion of it, we use a combination of regular expressions and an LLM-based autonomous inference. The regular expression, provided below, captures keywords indicating a need for a full-file analysis, such as "how many," "how much," "all," "whole," "total," "summary," and "entire." For example, if a user asks, "How many errors occurred?" or "Provide a total summary of events," the regular expression flags this as a query that likely requires examining the whole file.

In situations where a query targets specific sections of the log file, such as a recent time frame or a specific type of event, only a portion of the file needs to be evaluated. For instance, if a user asks, "Retrieve CPU usage data for the last 10 minutes" or "What is the status of the most recent log entry?" these queries require targeted information from particular segments rather than a comprehensive analysis. This approach allows for efficient processing by narrowing down the scope to relevant sections instead of analyzing the entire log.

The regular expression and prompt template are below:

```
Regular expression:
\b(\d{1,2}:\d{2}(:\d{2})?( ?(AM|PM))?|\b\d{4}-\d{2}-\d{2}\b|\b\d{2}/\d
   {2}/\d{4}\b|\btimestamp\b|\btime\b|\bdate\b)\b

Prompt Template:
Analyze each query to determine the necessary scope of log file evaluation
   ,
responding with Full Evaluation for complete file analysis or Portion
   Evaluation
for specific sections.
Examples:
Query: Identify all instances of error events throughout the log file.
Response: Full Evaluation
Query: Retrieve CPU usage data for the last 10 minutes of the log.
Response: Portion Evaluation
Query: Summarize service failures across the log file.
Response: Full Evaluation
Query: What is the status of the most recent log entry?
Response: Portion Evaluation
Template:
Query: [User Query Here]
Response:
```

6.3.2 Active Retrieval

This module is responsible to effectively analyze our log files based on the decision our decision-making module has been made. Below, we will explain each decision in detail:

Timestamp strategies

The strategy involves timestamp-based extraction. If a timestamp is explicitly mentioned, the decision-making module provides a specific timestamp to the active retrieval module, which then extracts logs from a 30-second window surrounding the specified time. Should the user manually select a start and end time, the module focuses on extracting logs within this user-defined range. In the absence of specific timestamps, the log file is chunked using a recursive character text splitter (e.g., through LangChain).

Portion of content evaluation

For scenarios where only a portion of the log file needs to be analyzed, we prioritize the most relevant chunks using an embedding-based technique and calculating the cosine similarity between the query of user and the log file chunks. Based on empirical evidence and trial-and-error optimization, we found that sending the top three most relevant chunks produces more accurate results without exceeding the token limits of the model. Considering our resource limitations, we also tested several lightweight embedding models, including distilBERT-base-uncased, BERT-base-uncased, intfloat/e5-base, and TinyBERT GENERAL 4L 312D. After manually evaluating 20 ground-truth examples through domain experts, TinyBERT GENERAL 4L 312D demonstrated better accuracy compared to the other models, so we chose this model for our embedding module.

Full content evaluation

For cases where the entire log file needs to be analyzed, we have two main strategies which are our main contributions.

1- Token Count Tracking and Summarization Strategy: For the first strategy, we keep track of the token limit. We start a new conversation for each log file the user wants to evaluate. If we hit the token limit, we summarize the previous conversation using text summarization of GPT-4o and for its prompt we used self-prompt technique. Then we send the summary received from LLM model along with the new chunk of the log file and the query of user to a new conversation. This way, we stay within the token limits while maintaining

the context of the analysis.

2- Flame-graph-like Strategy: For the second strategy, we create the **Flame-graph-like** structure to reduce the input tokens in our prompt while keeping most critical contents.

Flame graphs have become an essential tool in performance diagnostics by visually representing hierarchical patterns of system activities, making it easier to identify bottlenecks or recurring issues [30]. Inspired by this capability, we propose a method for compressing log data to optimize token usage when sending it to Large Language Models (LLMs) for analysis. The aim is to avoid exceeding token limits by using flame graph-like structures to reduce redundancy in the logs. This approach reduces token limitation keeping intact the relationships between logs and events. Instead of sending entire log files, which can be large and redundant, we group similar logs and send a flame graph representation that retains both contextual and temporal relationships. This method offers an efficient way to process log data while ensuring that relevant patterns are maintained for LLM-based analysis. An example of the flame-graph is shown in Figure 6.3

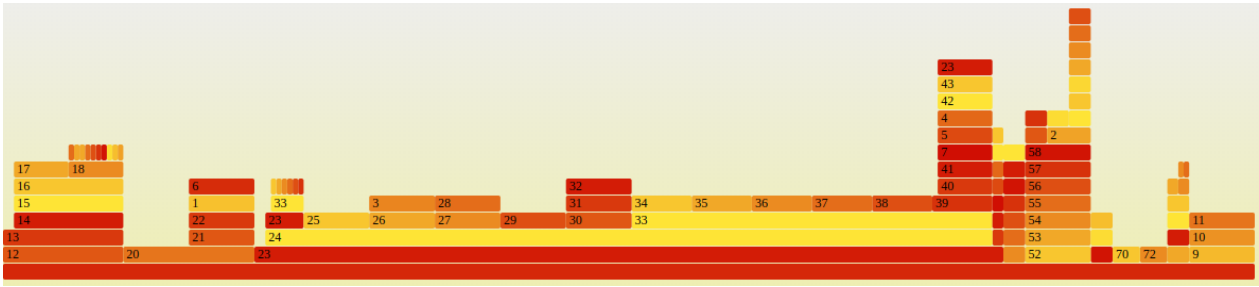


Figure 6.3 An example visualization of a flame graph for log sequences mapped to cosine similarity group numbers. The stacks represent log sequences, and the width of each stack indicates its occurrence frequency within a log file.

The flame-graph-like approach follows a multi-stage process, described below:

Step 1: Parsing and Grouping Log Messages by Similarity:
The process begins by parsing each log entry to extract the timestamp and message fields. To reduce redundancy, log messages are grouped based on their semantic similarity, using cosine similarity to measure how closely messages resemble each other. For example, consider a log file with entries like:

```
[2024-10-15 10:00:00.123] Error connecting to database
[2024-10-15 10:00:01.456] Error connecting to database
[2024-10-15 10:00:02.500] Failed to write to file
[2024-10-15 10:00:03.001] Error connecting to database
[2024-10-15 10:00:03.789] Successful login for user
```

```
[2024-10-15 10:00:05.000] Error connecting to database
```

This is achieved using cosine similarity, which compares the semantic similarity between log messages. In large log files, certain types of messages often recur frequently. For instance, error messages like "Error connecting to database" may appear multiple times. Using cosine similarity, we group these repeated or nearly identical logs together into clusters. This reduces redundancy by ensuring that only one representative log from each cluster is transmitted to the LLM. In effect, rather than sending multiple identical logs, the system sends a single compressed version of the message, drastically reducing the number of tokens required. This approach addresses the issue of large logs by focusing on the similarity of logs and minimizing duplication.

In this file, certain messages—such as "Error connecting to database"—occur multiple times. Cosine similarity groups these repeated or nearly identical messages into clusters, resulting in groups like:

```
Group 1: Error connecting to database (occurs 4 times)
Group 2: Failed to write to file (occurs 1 time)
Group 3: Successful login for user (occurs 1 time)
```

Step2: Creating a Time-Windowed Forest of Directed Acyclic Graphs (DAGs)
The log file is used to create directed trees of logs based on their timestamps, e.g If the next log is within the time-window of the last log, it is appended to that tree; otherwise it starts a new root stemming another tree. The process continues until we have parsed the complete log file. The algorithm can have many variations, e.g. the time window can be altered to cater to the physical problem.

```
Sequence 1: Root: Error connecting to database (10:00:00.123)
|-- Error connecting to database (10:00:01.456)

Sequence 2: Root: Failed to write to file (10:00:02.500)
|-- Error connecting to database (10:00:03.001)
|---- Successful login for user (10:00:03.789)

Sequence 3: Root: Error connecting to database (10:00:05.000)
```

Step3: Merge messages
In this step, each message is replaced by its assigned group number from Step 1. For each sequence, consecutive messages with the same group number are merged within the time window, and the start and end times of each merged message are recorded. This step resembles the flame graph approach by aggregating repetitive patterns into a single, compact representation that preserves the timing and structure of events. For example, for the sequence given

in Step 2, after replacing messages with group numbers and merging consecutive occurrences, the result will be:

```
Sequence 1: Group 1 (10:00:00.123 - 10:00:01.456)

Sequence 2: Root: Group 2 (10:00:02.500)
|-- Group 1 (10:00:03.001)
|---- Group 3 (10:00:03.789)

Sequence 3: Root: Group 1 (10:00:05.000)
```

By aggregating repeated log entries within each sequence and using time ranges, this step achieves a flame-graph-like structure, compressing each sequence into a concise format. This structure effectively reduces redundancy while maintaining essential information, allowing for efficient analysis and visualization. Prompt In the initial prompt, we send grouped log messages. Within each group, we only include the dynamic part of the log message, which is the actual message content. We discard static or redundant parts of the log, such as timestamps or repetitive metadata. By doing this, we effectively reduce the number of tokens used.

Additionally, we replace the full log messages in the flame-graph-like structure with their corresponding group numbers. It represents the messages based on the group they belong to. This further reduces the tokens while keeping essential information about the sequence and timing of events. The prompt result is something like as follows:

```
Prompt = {Query query} + file_content = Below is  a list of all the
   similar grouped messages from  the log file (don't refer to group
   number in your responses, instead refer to the specific logs that make
   up the group):

Grouped log messages:
    {
        "1": {
            "message": "Error connecting to database",
            "count": 4
        },
        "2": {
            "message": "Failed to write to file",
            "count": 1
        },
        "3": {
            "message": "Successful login for user",
            "count": 1
```

```
        }
    }

Below is a list of events which took place within the log file. Each
   number represents a log from that respective group and the timestamps
   show when the events took place (time of first and last log in the
   sequence)
The following event:

    a) 1 (10:00:00.123 - 10:00:01.456)
    b) 2 (10:00:02.500) -> 1 (10:00:03.001) -> 3 (10:00:03.789)
    c) 1 (10:00:05.000)
```

The flame graph above shows sequences of logs within specific time windows. This flame graph is different from traditional flame graphs, which come from sampled stack traces. Here is how to understand it:

- **Full Log Sequences:** Unlike regular, sample-based, flame graphs, this flame graph includes complete log sequences within each time window. There is no data loss.
- **Stacks Represent Log Sequences:** Each stack shows a sequence of logs in a time window. It follows the flow of events in that time frame.
- **Top of the Stack:** The top of each stack is the last log in the sequence. This shows where the sequence ends in the time window.
- **Stack Width and Frequency:** The width of a stack shows how many times that sequence appears in the logs. Wider stacks mean the sequence occurs more often, so common patterns stand out.
- **Forks Show Diverging Sequences:** Forks in the stack mean two log sequences meet at the same starting point (or root). This keeps important details in sub-sequences, so unique event paths remain visible.

This flame-graph-like approach compresses repeated log sequences and organizes them by time windows. It gives a simple but complete view of event patterns, which makes analysis easier and clearer.

6.3.3 Query Construction

In this Adaptive Framework, We used a self-promoting approach, inspiration from Jiao et al. [175], where the LLM model was asked to generate various prompts. Each prompt was then assessed on our dataset to identify the one that results the better responses from the LLM. We used a self-prompting strategy across all cases.

To efficiently process user queries in log analysis, we developed a tailored prompt framework

with several distinct types to address query needs. The framework works as follows:

— **Time-Specific Prompts:** If the user manually selects a specific time range, we send only that selected chunk. However, when a query explicitly mentions a time frame (e.g., between "1 PM and 1:15 PM"), an Explicit Time Prompt focuses on processing logs within that window.

— **Initial Chunk Evaluation Prompt:** For the initial processing, we use a few-shot prompt as a guideline for the model. This prompt explains the log file and its types, with examples tailored to the log file name and type. This information is sent along with the user query and the first chunk for evaluation, whether the task involves analyzing the whole file or selecting the top three chunks.

— **Extended Evaluation Prompt:** For subsequent chunks, we use a different strategy depending on whether the evaluation is for the whole file or partial file. For whole-file evaluation, we ask the model to aggregate its previous responses while ensuring uniqueness and eliminating redundancies. For partial evaluation, we use a specific prompt to guide the model in analyzing the relevant chunks individually. For example, if the question is "How many services are down?", the model analyzes the log file using parameters like module ID and status to identify shut-down services. It then combines relevant responses from previous chunks, ensuring no overlap or duplication in the final result.

— **Token Limit Management with Summarization:** When we are close to reaching the token limit for a conversation, we use two prompts to handle this condition. The first prompt summarizes all previous responses from previous conversations. The second prompt within a new conversation sends the summary of the previous conversation, user query, and new chunks of log.

— **Self-Assessment Hallucination Mitigation:** We created self-assessment prompts to guide the model in checking its own responses. These prompts are applied to each response we receive to reduce the chance of hallucinations and increase the likelihood of relevant answers.

— **System Prompt (Instructor Prompt):** To ensure accurate and well formatted responses, we carefully select the system prompt based on some empirical studies and user feedback. Additionally, to minimize the chance of hallucinations, we incorporated chain-of-thought prompting.

As part of our query construction strategy, for each prompt, we asked the LLM model to provide suggestions. We then limited our evaluation to a portion of our logs to choose which prompts produced better responses. Below, we demonstrate some of the suggested prompts. We applied them to different log types and evaluated the responses. By calculating the F1

score in Figure 6.4, we selected the prompts that performed better. We repeated this process for all of our prompts.

1. Using the information in the following content, which explains [brief explanation], please answer this query: [User Query]. Content: [Insert Content Here].
2. Given the details in the provided content, which describes [brief explanation], what is the answer to this question: [User Query]? Content: [Insert Content Here].
3. Please find the answer to the user's question based on the content below, which includes [brief explanation]: [User Query]. Content: [Insert Content Here].
4. Based on the content provided, which details [brief explanation], answer the following query: [User Query]. Content: [Insert Content Here].

To avoid bias in our self-prompting and self-assessment process, such as when an analyst concentrates only on one component of the system or looks at only one part of the logs rather than all relevant logs, we used structured and standardized prompts, thereby avoiding any personal judgment at all stages. We encouraged periodic peer analyses and validated results through cross-validation of multiple log sources to ensure reliable and unbiased findings.

6.3.4 Anonymization

To ensure that no critical data is leaked outside the system, we implemented an anonymization module that sanitizes both the log file and the user query. This module anonymizes sensitive data before sending the data to the LLM. After receiving the response from the model, the system reverts the anonymized data back to its original form to preserve the privacy of sensitive data throughout the entire process. We performed a reverse mapping to restore the anonymized items to their original form before displaying them to the user.

Based on a comprehensive survey of anonymization techniques [173], we anonymized sensitive information such as IP addresses, specific names, function names, module names, and directories. Based on the experience of our experts, we found that most user queries from logs focus on message content and timestamps. Since timestamps are not considered private data and messages are embedded within functions, this information is crucial for identifying service or function correlations and understanding the sequence of events.

Since this anonymization process can make it more difficult for the model to understand and learn from the data, we conducted an evaluation in Section 6.4 to have more accurate results from GPT-4o. For instance, random strings used in place of function names may not be easily interpreted by the model, which could impact the analysis. To solve this, we optimized our approach to make it understandable for the model and still having privacy. According

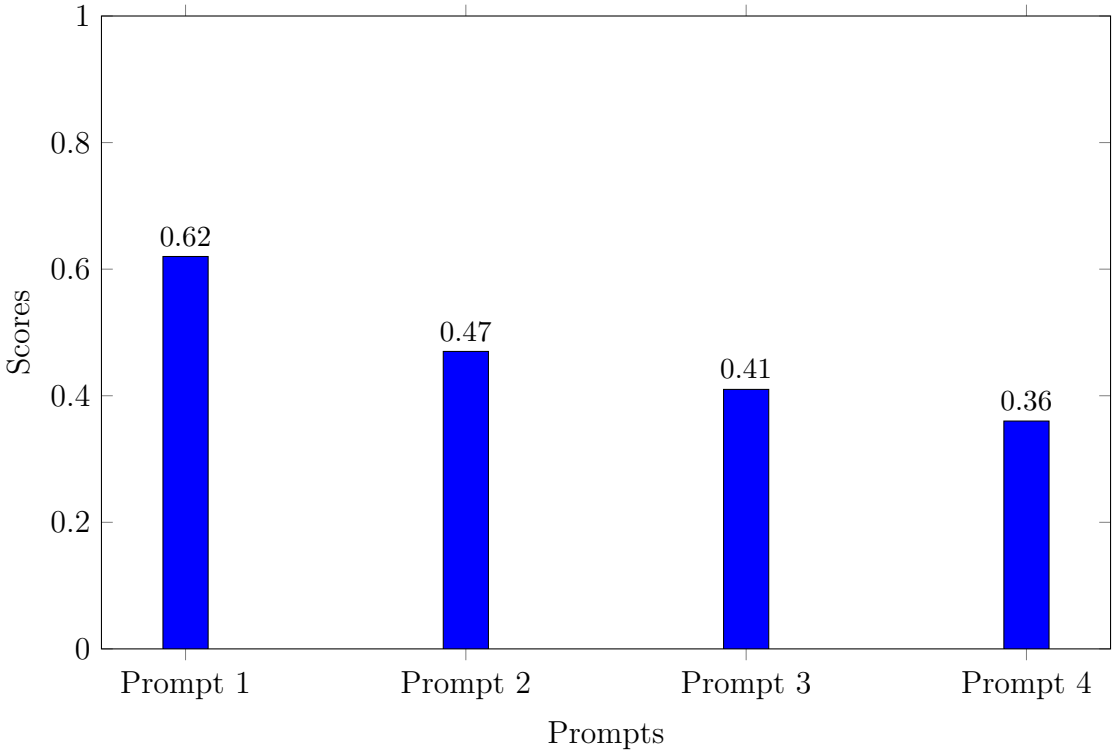


Figure 6.4 F1 Scores for Different Prompts in the self-prompt strategy.

to the privacy conditions of Ciena, we are unable to provide further details on the specific implementation of this module. However, we note that the techniques mentioned above work for any type of log with timestamps and messages.

6.3.5 Generation

In this module, we manage communication with the model by sending prompts generated from the query construction process. We keep track of the total token count—including both input tokens and model responses—within each conversation. When we approach the maximum token limit, we summarize the current conversation and initiate a new one to maintain context without exceeding limits. We evaluate the accuracy of the responses of the model through self-assessment strategies and integrate all responses from each chunk to form a complete and accurate answer. In case the response is inaccurate, the module will repeat asking from the active retrieval module for another chunk up to three times, which can be configured.

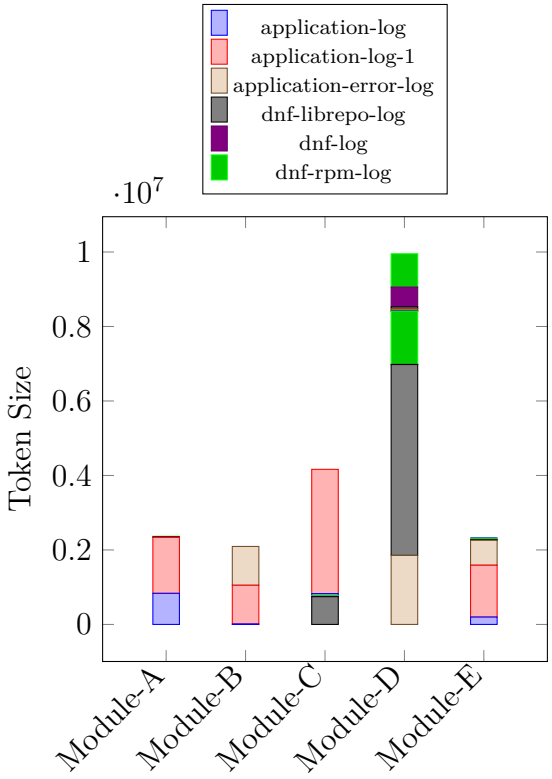


Figure 6.5 Stacked bar chart illustrating the total token sizes of various log files across different Module-E modules. The x-axis represents the modules—Module-A, Module-B, Module-C, Module-D, and Module-E—while the y-axis shows the total token size for each log. Each bar segment corresponds to a specific log file as indicated in the legend: application-log, application-log-1, application-error-log, dnf-librepo-log, dnf-log, and dnf-rpm-log. This visualization highlights the distribution and magnitude of log data within each module, highlighting the total size of different internal logs.

Anonymization	Precision (%)	Recall (%)	F1 Score (%)
RadomValue (Baseline)	86.67	23.64	37.26
FunctionNameRandomValue	**100.00**	**80.00**	**88.89**
FunctionName_RandomValue	100.00	76.36	86.57
FunctionName_RandomValue()	0.00	0.00	0.00
RandomValue()	100.00	27.27	42.86
functionNameRandomValue	100.00	27.27	42.86
RandomValue_FunctionName	86.67	20.00	32.47

Table 6.1 Summary of anonymization results showing precision, recall, and F1 score. **FunctionNameRandomValue** demonstrates the best performance, with an F1 score improvement of approximately 138.63% over the baseline.

6.4 Experimental Evaluation

In this section, we evaluate the contribution of our approaches, which include optimizing the context window size, implementing a flame-graph-like approach, and ensuring effective anonymization. We collected 20 questions with expected answers as ground truth from experts at Ciena, including developers, architects, maintenance engineers, and product owners. Due to internal legal and privacy policies at Ciena, we cannot present our datasets, which include internal logs and ground truth questions with their expected answers. Our server is equipped with a 10-core Intel Xeon processor, clocked at 2.8 GHz, with a 32 KB L1d cache, a 32 KB L1i cache, a 4 MB L2 cache, and a 16 MB L3 cache, alongside 59 GB of RAM. Our system comprises five main modules, each containing multiple services, totaling 65 log types with an average token size of 402,653 tokens, and an average log size of 1.5 GB with the range between 28 KB to 3 GB. The logs are stored in various formats, including JSON, text, CSV, and CTF, each with its own structure. However, all logs share common fields such as timestamps, line numbers, file names, function names, and message content illustrated in Figure 6.5. We examine how the context window size affects inference latency and accuracy, analyze the impact of total token size on the flame-graph-like approach, and assess the accuracy of our anonymization. For simplicity, we kept some parameters fixed while focusing on finding optimal values for the key ones.

6.4.1 The impact of the context windows size on inference accuracy and latency

In this study, due to limited computational resources, we selected lightweight embedding models, including distilBERT-base-uncased, BERT-base-uncased, intfloat/e5-base, and TinyBERT GENERAL 4L 312D. After evaluating 20 ground-truth examples, TinyBERT GEN-

Input-output Tokens	Input Tokens	Output Tokens	Latency (Sec.)	Total Latency (Sec.)	Precision
Test 500,500	207	85	2.086	16.378	3.3%
	225	86.5	2.103		
	244	79.5	2.155		
Test 1000,500	470	122.5	2.718	15.177	10%
	511.5	122	2.458		
	499.5	129	2.923		
Test 1500,500	739	152.5	3.202	14.425	10%
	725.5	106	3.59		
	724.5	106	2.956		
Test 2000,500	963	152.5	4.095	15.351	16.67%
	963	152.5	4.190		
	963.5	152.5	3.831		
Test 2500,500	1225	127.5	2.610	12.787	13.33%
	1223	125	3.429		
	1223	125	2.515		
Test 3000,500	1446	125	3.383	15.89	20%
	1447.5	125	5.43		
	1449	125	3.582		
Test 3500,500	1682.5	81.5	2.472	12.96	12%
	1668	81.5	2.472		
	1682	81.5	3.027		
Test 11000,500	2829.5	139.5	3.732	13.114	40%
	5380	148	4.068		
	5260	153.5	3.251		
Test 20000,500	9531.5	109.5	3.524	16.55	83.33%
	9530	109.5	4.707		
	7720.5	109.5	5.227		

Table 6.2 Latency measurements per API call and chunk relevance for different input token sizes along with the precision of the responses. The table illustrates how varying input token sizes (ranging from 500 to 20,000 tokens) affects the response latency of the OpenAI GPT-4 API and the overall latency of our pipeline when processing a log file of 15,869 tokens. It also presents the relevance of the top three selected chunks to user queries.

Name	Total Tokens	Flame-graph Tokens	Pipeline Latency (Sec.)	Flame-graph Latency (Sec.)
Log A	854 754	47 347	11.87	1.92
Log B	2179	278	8.94	0.63
Log C	1 517 318	32 772	18.44	3.67
Log D	770 823	141 728	13.65	6.55
Log E	8682	1994	2.41	0.04
Log F	1 762 090	61 854	18.13	3.91
Log G	4 764 844	177 248	27.37	10.96
Log H	1 819 004	16 074	16.75	3.91
Log I	6216	261	2.62	0.66

Table 6.3 This table presents data for Logs A to I, including the total token size, the reduced token count after applying the Flame-graph-like optimization, and the execution latencies in seconds for both the Pipeline and Flame-graph-like approach. The results highlight the relationship between token size and latency. It shows that the Flame-graph-like approach reduces both the token size by approximately 77% to 99.99% and execution latency by approximately 52% to 98% across all logs compared to the general Pipeline approach.

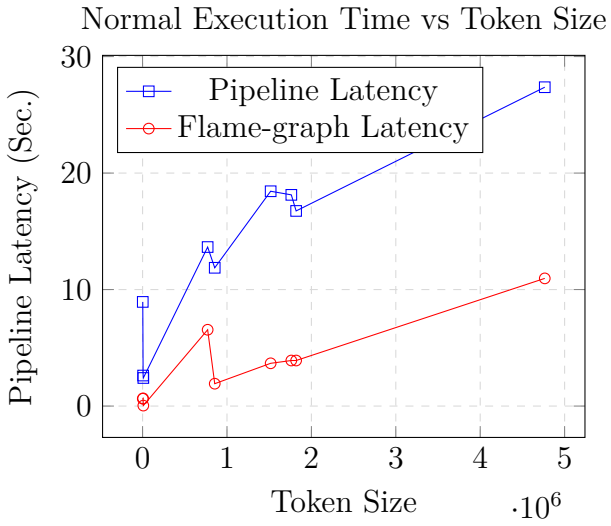


Figure 6.6 Execution latency versus token size for Pipeline and Flame-graph methods. The x-axis represents the token size, and the y-axis shows the execution latency in seconds. The chart demonstrates that as the token size increases, latency rises for both methods. However, the Flame-graph-like approach consistently exhibits lower latency compared to the Pipeline method across all token sizes. The Flame-graph-like approach achieves latency reductions, with improvements ranging from 52% to over 98% compared to the Pipeline method. Thus, the Flame-graph method is more efficient, reducing latency by over 90% at smaller token sizes and by around 60% at larger token sizes. This shows the effectiveness of Flame-graph over the general Pipeline method in processing logs more quickly, regardless of token size.

ERAL 4L 312D emerged as the most accurate model and was therefore chosen for the embedding module. To improve chunking efficiency, we employed the Recursive Character Text Splitter from LangChain. Since our application is in production mode and each token call has a cost, we set the output token limit to three to stay within budget. For consistency, we used two log types with an average total of 15,869 tokens in all evaluations and repeated the evaluation process 27 times to ensure reliable results.

To reduce the effect of pipeline latency, we focused not only on measuring the latency of each API call but also on evaluating the latency of the entire pipeline. This approach allowed us to see how different input token sizes affected the response time of OpenAI GPT-4o and the total latency of our basic pipeline. Table 6.2 presents detailed latency measurements for each API call, the overall pipeline latency, and the precision of the responses, allowing us to assess system performance. It shows latency per API call across token limits ranging from 500 to 20,000 tokens, including latency on individual api call and total pipeline response times. For example, tests with an 11,000-token input illustrate a trade-off: higher token limits increase latency but enhance the relevance of selected chunks. As shown in Figure 6.6, both per-call latency and overall pipeline latency exhibit an almost linear increase with the growth in token size. This shows the direct relationship between token size and latency. It emphasizes the importance of optimizing token usage to maintain system efficiency. To optimize this balance, we aim to find the best compromise between our budget per token call, the precision of results, and acceptable latency. In our evaluation, we found that an 11,000-token limit offered the best trade-off among these factors.

Lesson learned 1: A larger token limit led to higher latency and cost, but it also made the selected chunks more relevant. This shows a trade-off between performance and cost.

6.4.2 The impact of the log size on inference latency

We evaluated the effectiveness of a flame-graph-like approach in reducing token sizes when using LLMs as a service. For this, we applied the flame-graph-like approach to nine different logs and asked three questions for each log. Each query required a whole-file evaluation where the entire file was sent to the LLM for processing. We then measured the average latency, as shown in Table 6.3.

Additionally, we compared the latency of this flame-graph-like approach to a general pipeline solution. The latency, defined as the time from when a user submits a query to when the result is returned, was measured for both approaches across varying token sizes. As illustrated in Figure 6.6, the general pipeline showed significantly higher latency as token size increased, with a peak of 27.37 seconds for 4,764,844 tokens. In contrast, our flame-

graph-like solution demonstrated much more stable performance, maintaining lower latency and reaching a maximum of 10.96 seconds for the same token size.

By using our flame-graph-like approach, we achieved an average token reduction of 93.61% across all logs. This method also reduced processing latency by 77.45% compared to the general pipeline. it shows efficiency in optimizing LLM-based processing for large files.

This substantial improvement in latency shows the efficiency of flame-graph in reducing the token size and processing time. It makes a valuable optimization for LLM-based services that are sensitive to token-related costs.

Lesson learned 2: The flame-graph-like approach effectively reduces token size and processing time, achieving substantial reductions in both latency and token usage compared to standard methods. This makes it a useful way to optimize applications that need faster processing and lower token costs.

6.4.3 The impact of the anonymization and accuracy

In this module, we evaluate the effectiveness of customized anonymization compared to random mapping for private entities. Based on our expert experience, function names are the most critical data points, as they help us track events in messages, with each message originating from a specific function. Therefore, we focused on anonymizing function names specifically. Due to page limitations, we have not included the same procedure for other sensitive data, although the approach would be the same as explained here. We used the prompt below to evaluate this customized mapping and applied it across 10 of our log files.

```
Prompt: List all the unique function names in this log file.
```

As the average result shown in Table 6.1, random mapping can sometimes reduce the ability of the module to make accurate inferences. However, our results show that adding "FunctionName" as a prefix to the randomly generated names improved module performance, providing better accuracy than other approaches.

Lesson learned 3: Applying structured prefixes and suffixes to anonymized entities can enhance model performance and accuracy in large language models by keeping important details in sensitive data. In our case, appending "FunctionName" as a prefix to anonymized function names improves model performance and accuracy in anonymized data processing.

6.5 Limitations

This experience report introduces a novel approach to automating log analysis that effectively reduces token usage and enhances anonymization. Despite its potential to improve system reliability and reduce maintenance costs, several limitations must be acknowledged:

— In this project, we faced several resource limitations, particularly regarding computational resources like GPU and memory. As a result, we were constrained to use lightweight embedding vectors, which led to reduced accuracy in retrieving the most relevant chunks.

— Due to budget limitations associated with token calls, we limited our parameter tuning process to the top three chunks. We recommend that future researchers consider adjusting these limitations in their own applied projects to achieve more accurate results.

— Due to the internal policies of the company and privacy concerns, GPT-4o is the designated model, limiting access to other models. However, in the future, we may be able to evaluate additional models for comparison.

Future researchers are highly encouraged to perform comparisons between models, including open-source LLMs, and to apply the proposed pipeline and flame-graph-like approach to various datasets. Additionally, while each dataset has its own unique privacy considerations, the general anonymization approach outlined here can be adapted to meet specific privacy needs.

6.6 Conclusion

Many companies face challenges when processing and analyzing large amounts of log data. Software logs are dynamic and constantly changing, making it difficult to evaluate them manually. For anomaly detection or any other investigation, large language models (LLMs) can help automate this process. However, due to token limits and high computational demands, handling very large logs in-house remains challenging. Given these constraints, it is often more practical for companies to rely on external LLM models. However, using LLMs for log analysis can be costly, and data privacy is a concern, especially when relying on LLM as a service.

In this industrial experience, we proposed a flame-graph-like approach to reduce the input tokens required for log analysis. Our method achieved an average reduction of 93.61% in input tokens across all tested logs, effectively overcoming token limits. This approach also enhanced computational efficiency, reducing processing time by 77.45% compared to the basic

pipeline.

Our dynamic pipeline adapts to user queries, whether they need insights from a specific log segment, the entire log file, or a particular time frame. We also proposed a solution to anonymize sensitive data in logs, which improved response accuracy and reduced hallucinations compared to general mapping methods.

Each stage of our pipeline provided valuable insights that we believe will be useful for future researchers and industry practitioners.

Acknowledgment

This work is supported by Ciena. We would also like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson, and EfficiOS for supporting this research. We also thank Prince Kapoor and our colleagues at Ciena for their collaboration in this project.

CHAPITRE 7 CONCLUSION

Cette section présente une synthèse des travaux réalisés dans le cadre de ce projet de recherche. La thèse se conclut par des perspectives de recherche futures pouvant s'appuyer sur les contributions proposées.

7.1 Synthèse des travaux

Les deux nouvelles méthodes présentées dans le Chapitre 4 apportent des améliorations significatives au diagnostic des problèmes de performance dans les systèmes distribués. La première méthode, une visualisation de graphe de flammes différentiel, permet une comparaison intuitive et évolutive entre des ensembles de traces, en agrégeant et en colorant les variations de performance au niveau des intervalles (spans). Cela aide à identifier les régressions ou optimisations entre versions ou configurations. La seconde méthode, un module d'extraction de métriques au niveau noyau, fournit une vision détaillée du comportement système, incluant les attentes d'E/S, les contentions CPU, et la durée des appels système, permettant une analyse des causes racines au-delà des capacités des outils de traçage traditionnels. Ces deux méthodes forment une solution complète qui lie traçage haut niveau et métriques bas niveau. L'évaluation a montré qu'elles introduisent une surcharge minimale (inférieure à 9 % en mode instantané, et 10 % en mode standard), les rendant utilisables en production.

Le Chapitre 5 présente deux techniques complémentaires qui améliorent la précision et l'efficacité de l'analyse des causes racines dans les systèmes à microservices distribués. La première technique repose sur une stratégie de traçage hybride sensible au chemin critique, focalisant l'analyse sur les intervalles impactant directement la latence de bout en bout. Cette focalisation réduit la charge de calcul et permet un diagnostic plus ciblé. La seconde technique intègre la technique Personalized PageRank avec la localisation de fautes basée sur le spectre, améliorant la précision tout en minimisant la consommation de stockage, en collectant des métriques uniquement pour les intervalles les plus pertinents. Ensemble, ces techniques réduisent de plus de 22 % le nombre d'intervalles à analyser et diminuent de 99 % les métriques noyau stockées, rendant l'approche évolutive et adaptée aux environnements réels.

Enfin, afin de faire gagner du temps aux équipes industrielles utilisant actuellement une analyse manuelle des journaux, une approche automatisée basée sur les modèles de langage (LLMs) a été proposée. Le Chapitre 4 introduit deux techniques clés pour surmonter

les limites actuelles des LLMs appliqués à de grands fichiers de logs. La première est une stratégie adaptative qui détermine s'il faut analyser l'ensemble du journal, une période spécifique, ou seulement les segments les plus pertinents. Cela permet de réduire le nombre de jetons utilisés et la latence des réponses. La seconde technique repose sur une méthode de compression inspirée des graphes de flammes, qui regroupe les entrées similaires dans des séquences temporelles. Cette méthode réduit la taille des entrées jusqu'à 93.61 %, tout en maintenant le contexte nécessaire à l'analyse. Ces deux techniques offrent une solution évolutive, respectueuse de la confidentialité, et adaptée à l'analyse automatisée de logs en contexte industriel.

7.2 Limitations

Bien que les trois approches proposées dans les Chapitres 4, 5, 6 — DTraComp, l'analyse hybride sensible au chemin critique, et InsightAI — représentent des avancées notables en matière d'observabilité et de diagnostic, elles présentent également certaines limitations.

DTraComp (Chapitre 4) permet une comparaison efficace des traces distribuées via les graphes de flammes différentiels, mais comporte plusieurs limites. Chaque machine dans un système distribué possède sa propre horloge, ce qui peut introduire de légers décalages temporels. Trace Compass ajuste partiellement les horodatages via les messages réseau, mais une synchronisation parfaite reste difficile. Ces écarts peuvent impacter l'analyse fine. De plus, reproduire les expériences sur d'autres systèmes est complexe, notamment en l'absence de détails sur la configuration (matériel, version logicielle, paramètres réseau). DTraComp requiert également un double niveau de traçage (haut et bas niveau), ce qui complique la mise en œuvre.

L'outil est optimal lorsqu'il compare deux groupes de requêtes similaires. Il détecte moins bien les anomalies rares ou atypiques, hors de la plage sélectionnée. Les résultats dépendent fortement de la qualité des données de trace collectées — des événements manquants ou une mauvaise configuration peuvent mener à de fausses conclusions. Par ailleurs, l'outil repose sur des visualisations complexes (graphes de flammes) qui nécessitent une expertise préalable. Enfin, bien que conçue pour minimiser la surcharge, l'analyse de traces détaillées peut affecter les performances globales sur des systèmes à forte charge.

L'approche du Chapitre 5 est légère et efficace pour la détection de causes racines, mais elle présente plusieurs limites. Elle se concentre uniquement sur le chemin critique, ce qui peut omettre des problèmes en dehors de celui-ci. Elle fonctionne mieux lorsque la majorité des requêtes sont normales, ce qui n'est pas toujours le cas. La reproduction de certaines

méthodes depuis la littérature, faute de code source, peut introduire des biais. Le recours à des correspondances exactes pour classifier les requêtes améliore l'efficacité, mais limite la détection d'anomalies dans les requêtes légèrement différentes. La synchronisation d'horloges entre composants distribués reste un défi, et les évaluations sont réalisées principalement dans des environnements contrôlés.

L'approche du Chapitre 6, basée sur les LLMs, permet une analyse efficace de logs à grande échelle, mais présente également des limites. L'infrastructure utilisée ne disposait pas de matériel puissant (GPU, mémoire), ce qui a nécessité l'utilisation de modèles légers, moins précis. Pour réduire les coûts, seuls les trois segments les plus pertinents étaient analysés, ce qui peut laisser passer des informations importantes. De plus, les contraintes d'entreprise ont imposé l'usage exclusif de GPT-4o, empêchant l'évaluation d'autres modèles plus performants ou respectueux de la vie privée.

7.3 Perspectives de recherche

Les travaux futurs peuvent viser à rendre DTraComp (Chapitre 4) plus utile et plus facilement déployable dans des systèmes réels. Une piste serait d'améliorer la synchronisation des horloges entre les machines, afin d'augmenter la précision. Une autre direction est de permettre la comparaison de plusieurs groupes de requêtes, au-delà des comparaisons binaires actuelles. Il est aussi possible d'améliorer le regroupement des types de requêtes, afin d'intégrer des requêtes ayant des structures légèrement différentes. Réduire la complexité de l'instrumentation et la configuration du traçage serait également bénéfique. Enfin, des expérimentations sur des systèmes distribués de grande taille permettraient d'évaluer la scalabilité et la performance en environnement de production. L'intégration avec des outils d'analyse automatique ou de machine learning pourrait aussi améliorer le diagnostic.

Concernant le Chapitre 5, les recherches futures peuvent viser la détection d'anomalies hors du chemin critique, notamment dans les services en arrière-plan ou en parallèle. Une autre piste est l'adaptation à des environnements où les requêtes anormales sont plus nombreuses que les requêtes normales. L'apprentissage automatique de la structure des requêtes, en remplacement de l'appariement exact, permettrait de gérer davantage de variations. Améliorer la synchronisation entre machines pour le traçage hybride renforcerait également la précision. Tester la méthode sur des systèmes en production à grande échelle, et comparer plus de modèles, fourniraient des résultats plus représentatifs.

Enfin, les travaux futurs relatifs au Chapitre 6 peuvent explorer l'utilisation d'autres modèles de langage, notamment en code source libre, afin de comparer la précision, le coût et

la protection de la vie privée. L'amélioration de la sélection des segments log les plus pertinents, via de meilleurs modèles d'intégration (embedding models), est une piste prometteuse. Une meilleure gestion des limites de jetons, par des méthodes de résumé ou de sélection intelligente, pourrait également améliorer les performances. L'approche actuelle, analysant seulement trois segments, peut être étendue à une analyse complète du journal si nécessaire. Enfin, la méthode d'anonymisation pourrait être testée et renforcée dans des contextes de protection stricte des données.

RÉFÉRENCES

[1] Tracecompare, enhanced differential flame graph. [Online]. Available: https://tracingsummit.org/ts/2015/files/TracingSummit2015-TraceCompare.pdf

[2] Y. Song, "Tidb: The performance drop 70 percent," https://github.com/pingcap/tidb/issues/44584, 2023.

[3] F. F. Daneshgar, M. Dagenais, M. Ekhlasi *et al.*, "Dtracomp: Comparing distributed execution traces for understanding intermittent latency sources," *Authorea*, October 18 2023.

[4] Y. Tan and X. Gu, "On predictability of system anomalies in real world," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 2010, pp. 133–140.

[5] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 452–461.

[6] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. Ward, "System monitoring with metric-correlation models: problems and solutions," in *Proceedings of the 6th international conference on Autonomic computing*, 2009, pp. 13–22.

[7] G. Jiang, H. Chen, and K. Yoshihira, "Modeling and tracking of transaction flow dynamics for fault detection in complex systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 4, pp. 312–326, 2006.

[8] M. A. Munawar and P. A. Ward, "A comparative study of pairwise regression techniques for problem determination," in *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, 2007, pp. 152–166.

[9] ——, "Leveraging many simple statistical models to adaptively monitor software systems," in *Parallel and Distributed Processing and Applications: 5th International Symposium, ISPA 2007 Niagara Falls, Canada, August 29-31, 2007 Proceedings 5*. Springer, 2007, pp. 457–470.

[10] K. Shen, C. Stewart, C. Li, and X. Li, "Reference-driven performance anomaly identification," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1, pp. 85–96, 2009.

[11] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan, "Online detection of utility cloud anomalies using metric distributions," in *2010 IEEE Network Operations and*

Management Symposium-NOMS 2010. IEEE, 2010, pp. 96–103.

[12] S. Ghosh, A. Hamou-Lhadj, and N. Ezzati-Jivan, "System and application performance analysis patterns using software tracing," Ph.D. dissertation, Concordia University, 2022.

[13] What is debugger? [Online]. Available: https://en.wikipedia.org/wiki/Debugger

[14] Debugging vs. testing. [Online]. Available: https://www.techtarget.com/searchsoftwarequality/definition/debugging

[15] S. Shende, "Profiling and tracing in linux," in *Proceedings of the Extreme Linux Workshop*, vol. 2, 1999.

[16] P.-M. Fournier and M. R. Dagenais, "Analyzing blocking to debug performance problems on multi-core systems," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 77–87, 2010.

[17] D. Narayanan, "End-to-end tracing considered essential," in *Proceedings of High Performance Transaction Systems–Eleventh Biennial Workshop (HPTS'05)*, 2005.

[18] C. Cassé, P. Berthou, P. Owezarski, and S. Josset, "A tracing based model to identify bottlenecks in physically distributed applications," in *2022 International Conference on Information Networking (ICOIN)*, 2022, pp. 226–231.

[19] D. Narayanan, "End-to-end tracing considered essential," April 2005, proceedings of High Performance Transaction Systems - Eleventh Biennial Workshop (HPTS '05). [Online]. Available: https://www.microsoft.com/en-us/research/publication/end-to-end-tracing-considered-essential/

[20] M. Desnoyers and M. Dagenais, "Low disturbance embedded system tracing with linux trace toolkit next generation," in *ELC (Embedded Linux Conference)*, vol. 2006, 2006.

[21] D. Toupin, "Using tracing to diagnose or monitor systems," *IEEE Software*, vol. 28, no. 1, pp. 87–91, 2011.

[22] What is ebpf. [Online]. Available: https://ebpf.io/what-is-ebpf/

[23] What is perf. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[24] Understanding how systemtap works. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/systemtap_beginners_guide/understanding-how-systemtap-works

[25] Systemtap. [Online]. Available: https://en.wikipedia.org/wiki/SystemTap

[26] Opencensus. [Online]. Available: https://opentracing.io/

[27] Open tracing. [Online]. Available: https://opencensus.io/

[28] G. Leffler, "{OpenTelemetry} and observability: What, why, and why now?" 2022.

[29] Y. Shkuro, *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd, 2019.

[30] B. Gregg, "The flame graph," *Communications of the ACM*, vol. 59, no. 6, pp. 48–57, 2016.

[31] Differential flame graphs. [Online]. Available: https://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html

[32] C. H. Kim, J. Rhee, H. Zhang, N. Arora, G. Jiang, X. Zhang, and D. Xu, "Intropert: Transparent context-sensitive multi-layer performance inference using system stack traces," *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1, pp. 235–247, 2014.

[33] F. Doray and M. Dagenais, "Diagnosing performance variations by comparing multi-level execution traces," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 462–474, 2016.

[34] Trace compass. [Online]. Available: https://www.eclipse.org/tracecompass/

[35] Eclipse theia. [Online]. Available: https://theia-ide.org/docs/

[36] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance anomaly detection and bottleneck identification," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–35, 2015.

[37] H. Chen, G. Jiang, H. Zhang, and K. Yoshihira, "Boosting the performance of computing systems through adaptive configuration tuning," in *Proceedings of the 2009 ACM symposium on Applied Computing*, 2009, pp. 1045–1049.

[38] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[39] P. Chen, Y. Qi, P. Zheng, and D. Hou, "Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 2014, pp. 1887–1895.

[40] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer, "Sieve: Actionable insights from monitored metrics in distributed systems," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, 2017, pp. 14–27.

[41] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Service-Oriented Computing: 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings 16*. Springer, 2018, pp. 3–20.

[42] G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li, "Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments," in *Proceedings of the Web Conference 2021*, ser. WWW '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3087–3098. [Online]. Available: https://doi.org/10.1145/3442381.3449905

[43] Z. Chen, J. Liu, Y. Su, H. Zhang, X. Ling, Y. Yang, and M. R. Lyu, "Adaptive performance anomaly detection for online service systems via pattern sketching," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 61–72.

[44] J. Huang, B. Mozafari, and T. F. Wenisch, "Statistical analysis of latency through semantic profiling," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 64–79.

[45] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue *et al.*, "Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 48–58.

[46] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection from system tracing data using multimodal deep learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 179–186.

[47] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 807–817.

[48] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun *et al.*, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs." in *IJCAI*, vol. 19, no. 7, 2019, pp. 4739–4745.

[49] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 1285–1298.

[50] C. Liu, S. Ghosal, Z. Jiang, and S. Sarkar, "An unsupervised spatiotemporal graphical modeling approach to anomaly detection in distributed cps," in *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2016, pp. 1–10.

[51] C. Liu, K. G. Lore, and S. Sarkar, "Data-driven root-cause analysis for distributed system anomalies," in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE, 2017, pp. 5745–5750.

[52] G. Li, T. Yuan, S. J. Qin, and T. Chai, "Dynamic time warping based causality analysis for root-cause diagnosis of nonstationary fault processes," *IFAC-PapersOnLine*, vol. 48, no. 8, pp. 1288–1293, 2015.

[53] V. Cortellessa and L. Traini, "Detecting latency degradation patterns in service-based systems," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 161–172.

[54] M. Dymshits, B. Myara, and D. Tolpin, "Process monitoring on sequences of system call count vectors," in *2017 International Carnahan Conference on Security Technology (ICCST)*. IEEE, 2017, pp. 1–5.

[55] Q. Fournier, N. Ezzati-Jivan, D. Aloise, and M. R. Dagenais, "Automatic cause detection of performance problems in web applications," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 398–405.

[56] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.

[57] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[58] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[59] Q. Dong, L. Li, D. Dai, C. Zheng, J. Ma, R. Li, H. Xia, J. Xu, Z. Wu, T. Liu *et al.*, "A survey on in-context learning," *arXiv preprint arXiv:2301.00234*, 2022.

[60] T. Sun, Y. Shao, H. Qian, X. Huang, and X. Qiu, "Black-box tuning for language-model-as-a-service," in *International Conference on Machine Learning*. PMLR, 2022, pp. 20 841–20 855.

[61] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[62] X. Huang, T. Zhang, and W. Zhao, "Logrules: Enhancing log analysis capability of large language models through rules," in *Findings of the Association for Computational Linguistics: NAACL 2025*, 2025, pp. 452–470.

[63] 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.

[64] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, 2020, pp. 38–45.

[65] H. Jin, X. Han, J. Yang, Z. Jiang, Z. Liu, C.-Y. Chang, H. Chen, and X. Hu, "Llm maybe longlm: Self-extend llm context window without tuning," *arXiv preprint arXiv:2401.01325*, 2024.

[66] G. Jiang, H. Chen, and K. Yoshihira, "Discovering likely invariants of distributed transaction systems for autonomic system management," in *2006 IEEE International Conference on Autonomic Computing*. IEEE, 2006, pp. 199–208.

[67] S. Chauhan and L. Vig, "Anomaly detection in ecg time signals via deep long short-term memory networks," in *2015 IEEE international conference on data science and advanced analytics (DSAA)*. IEEE, 2015, pp. 1–7.

[68] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, and G. Shroff, "Lstm-based encoder-decoder for multi-sensor anomaly detection," *arXiv preprint arXiv:1607.00148*, 2016.

[69] P. Malhotra, L. Vig, G. Shroff, P. Agarwal *et al.*, "Long short term memory networks for anomaly detection in time series." in *ESANN*, vol. 2015, 2015, p. 89.

[70] S. Maya, K. Ueno, and T. Nishikawa, "dlstm: a new approach for anomaly detection using deep learning with delayed prediction," *International Journal of Data Science and Analytics*, vol. 8, pp. 137–164, 2019.

[71] Ú. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz, "Fay: Extensible distributed tracing from kernels to clusters," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, pp. 1–35, 2012.

[72] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling." in *OSDI*, vol. 4, 2004, pp. 18–18.

[73] A. Traeger, I. Deras, and E. Zadok, "Darc: Dynamic analysis of root causes of latency distributions," in *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2008, pp. 277–288.

[74] X. Gu and H. Wang, "Online anomaly prediction for robust cluster systems," in *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 2009, pp. 1000–1011.

[75] S. Huang, C. Fung, K. Wang, P. Pei, Z. Luan, and D. Qian, "Using recurrent neural networks toward black-box system anomaly prediction," in *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*. IEEE, 2016, pp. 1–10.

[76] A. W. Williams, S. M. Pertet, and P. Narasimhan, "Tiresias: Black-box failure prediction in distributed systems," in *2007 IEEE international parallel and distributed processing symposium*. IEEE, 2007, pp. 1–8.

[77] D. J. Dean, H. Nguyen, and X. Gu, "Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *Proceedings of the 9th international conference on Autonomic computing*, 2012, pp. 191–200.

[78] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, "Overload control for scaling wechat microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 149–161.

[79] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *2019 IEEE International Conference on Web Services (ICWS)*. IEEE, 2019, pp. 68–75.

[80] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 3–18.

[81] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2021.

[82] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 683–694.

[83] L. Gelle, N. Ezzati-Jivan, and M. R. Dagenais, "Combining distributed and kernel tracing for performance analysis of cloud applications," *Electronics*, vol. 10, no. 21, p. 2610, 2021.

[84] T. Davidson, E. Wall, and J. Mace, "A qualitative interview study of distributed tracing visualisation: A characterisation of challenges and opportunities," *IEEE Transactions on Visualization and Computer Graphics*, 2023.

[85] G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li, "Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments," in *Proceedings of the Web Conference 2021*, 2021, pp. 3087–3098.

[86] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, and M. D. Ernst, "Visualizing distributed system executions," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 2, pp. 1–38, 2020.

[87] V. Anand, M. Stolet, T. Davidson, I. Beschastnikh, T. Munzner, and J. Mace, "Aggregate-driven trace visualizations for performance debugging," *arXiv preprint arXiv:2010.13681*, 2020.

[88] L. Traini, J. Leone, G. Stilo, and A. Di Marco, "Vamp: Visual analytics for microservices performance," in *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, 2024, pp. 1209–1218.

[89] F. Stephanie, "A sense of self for unix processes," in *IEEE Symposium on Security and Privacy, 1996*, 1996, pp. 120–128.

[90] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *Proceedings of the 1999 IEEE symposium on security and privacy (Cat. No. 99CB36344)*. IEEE, 1999, pp. 133–145.

[91] "Sysdig: Security tools for containers, kubernetes, and cloud." [Online]. Available: https://sysdig.com/

[92] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

[93] A. Montplaisir-Gonçalves, N. Ezzati-Jivan, F. Wininger, and M. R. Dagenais, "State history tree: an incremental disk-based data structure for very large interval data," in *2013 International Conference on Social Computing*. IEEE, 2013, pp. 716–724.

[94] N. Ezzati-Jivan and M. R. Dagenais, "A framework to compute statistics of system parameters from very large trace files," *ACM SIGOPS Operating Systems Review*, vol. 47, no. 1, pp. 43–54, 2013.

[95] I. Kohyarnejadfard, D. Aloise, S. V. Azhari, and M. R. Dagenais, "Anomaly detection in microservice environments using distributed tracing data analysis and nlp," *Journal of Cloud Computing*, vol. 11, no. 1, pp. 1–16, 2022.

[96] C.-P. Bezemer, J. Pouwelse, and B. Gregg, "Understanding software performance regressions using differential flame graphs," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 535–539.

[97] R. R. Sambasivan, I. Shafer, M. L. Mazurek, and G. R. Ganger, "Visualizing request-flow comparison to aid performance diagnosis in distributed systems," *IEEE transactions on visualization and computer graphics*, vol. 19, no. 12, pp. 2466–2475, 2013.

[98] B. Poirier, R. Roy, and M. Dagenais, "Accurate offline synchronization of distributed traces using kernel-level events," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 75–87, 2010.

[99] H. Marouani and M. R. Dagenais, "Internal clock drift estimation in computer clusters," *Journal of Computer Systems, Networks, and Communications*, vol. 2008, 2008.

[100] M. Jabbarifar, M. Dagenais, and A. Shameli-Sendi, "Online incremental clock synchronization," *Journal of Network and Systems Management*, vol. 23, pp. 1034–1066, 2015.

[101] M. S. Dickson, "Distributed deadlock and distributed recovery," *Equatorial Journal of Communication Technology*, vol. 3, no. 02, pp. 20–24, 2020.

[102] E. Bauer and R. Adams, *Reliability and availability of cloud computing*. John Wiley & Sons, 2012.

[103] L. Wang, R. A. Hosn, and C. Tang, "Remediating overload in over-subscribed computing environments," in *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 2012, pp. 860–867.

[104] S. A. Baset, L. Wang, and C. Tang, "Towards an understanding of oversubscription in cloud," in *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12)*, 2012.

[105] D. Cotroneo, R. Natella, and S. Rosiello, "Overload control for virtual network functions under cpu contention," *Future Generation Computer Systems*, vol. 99, pp. 164–176, 2019.

[106] EclipseSource. (2024) Eclipse theia 1.46 release - news and noteworthy. Accessed: September 22, 2024. [Online]. Available: https://eclipsesource.com/blogs/2024/02/09/eclipse-theia-1-46-release-news-and-noteworthy/

[107] PingCAP. (2024) Investigate issue with INT_COL in tidb - github issue #44584. Accessed: September 22, 2024. [Online]. Available: https://github.com/pingcap/tidb/issues/44584

[108] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, "Tandem: A taxonomy and a dataset of real-world performance bugs," *IEEE Access*, vol. 8, pp. 107 214–107 228, 2020.

[109] Y. Meng, S. Zhang, Y. Sun, R. Zhang, Z. Hu, Y. Zhang, C. Jia, Z. Wang, and D. Pei, "Localizing failure root causes in a microservice through causality inference," in *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. IEEE, 2020, pp. 1–10.

[110] J. Xu, Z. Cui, Y. Zhao, X. Zhang, S. He, P. He, L. Li, Y. Kang, Q. Lin, Y. Dang *et al.*, "Unilog: Automatic logging via llm and in-context learning," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[111] B. Chen and Z. M. Jiang, "A survey of software log instrumentation," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–34, 2021.

[112] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A survey on automated log analysis for reliability engineering," *ACM computing surveys (CSUR)*, vol. 54, no. 6, pp. 1–37, 2021.

[113] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[114] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018.

[115] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs, *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020.

[116] A. Ezaz, G. Khodabandeh, and N. Ezzati-Jivan, "Analyzing performance variability in alibaba's microservice architecture: A critical-path-based perspective," in *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, 2024, pp. 82–86.

[117] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "Microrca: Root cause localization of performance issues in microservices," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.

[118] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng, "Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 553–565.

[119] L. Zheng, Z. Chen, D. Wang, C. Deng, R. Matsuoka, and H. Chen, "Lemma-rca: A large multi-modal multi-domain dataset for root cause analysis," *arXiv preprint arXiv:2406.05375*, 2024.

[120] A. Fang, "Root cause analysis for distributed systems."

[121] D. Wang, Z. Chen, J. Ni, L. Tong, Z. Wang, Y. Fu, and H. Chen, "Interdependent causal networks for root cause localization," *Proceedings of the 29th ACM SIGKDD*

Conference on Knowledge Discovery and Data Mining, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:260499433

[122] X. Zhang, Y. Xu, S. Qin, S. He, B. Qiao, Z. Li, H. Zhang, X. Li, Y. Dang, Q. Lin *et al.*, "Onion: identifying incident-indicating logs for cloud systems," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1253–1263.

[123] S. Gu, G. Rong, T. Ren, H. Zhang, H. Shen, Y. Yu, X. Li, J. Ouyang, and C. Chen, "Trinityrcl: Multi-granular and code-level root cause localization using multiple types of telemetry data in microservice systems," *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 3071–3088, 2023.

[124] Z. Li, N. Zhao, M. Li, X. Lu, L. Wang, D. Chang, X. Nie, L. Cao, W. Zhang, K. Sui *et al.*, "Actionable and interpretable fault localization for recurring failures in online service systems," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 996–1008.

[125] G. Yu, P. Chen, Z. He, Q. Yan, Y. Luo, F. Li, and Z. Zheng, "Changerca: Finding root causes from software changes in large online systems," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 24–46, 2024.

[126] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen *et al.*, "Automatic root cause analysis via large language models for cloud incidents," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 674–688.

[127] H. Wang, Z. Wu, H. Jiang, Y. Huang, J. Wang, S. Kopru, and T. Xie, "Groot: An event-graph-based approach for root cause analysis in industrial settings," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 419–429.

[128] L. Zheng, Z. Chen, J. He, and H. Chen, "Mulan: Multi-modal causal structure learning and root cause analysis for microservice systems," in *Proceedings of the ACM on Web Conference 2024*, 2024, pp. 4107–4116.

[129] C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu, "Eadro: An end-to-end troubleshooting framework for microservices on multi-source data," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1750–1762.

[130] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.

[131] Y.-Y. M. Chen, *Path-based failure and evolution management*. University of California, Berkeley, 2004.

[132] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi *et al.*, "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 34–50.

[133] Z. Zhu, C. Lee, X. Tang, and P. He, "Hemirca: Fine-grained root cause analysis for microservices with heterogeneous data sources," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–25, 2024.

[134] P. Chen, Y. Qi, and D. Hou, "Causeinfer: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment," *IEEE transactions on services computing*, vol. 12, no. 2, pp. 214–230, 2016.

[135] B. Eaton, J. Stewart, J. Tedesco, and N. C. Tas, "Distributed latency profiling through critical path tracing: Cpt can provide actionable and precise latency analysis." *Queue*, vol. 20, no. 1, pp. 40–79, 2022.

[136] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[137] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th international conference on Software engineering*, 2002, pp. 467–477.

[138] J. Jiang, R. Wang, Y. Xiong, X. Chen, and L. Zhang, "Combining spectrum-based fault localization and statistical debugging: An empirical study," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 502–514.

[139] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[140] G. Yu, Z. Huang, and P. Chen, "Tracerank: Abnormal service localization with disaggregated end-to-end tracing data in cloud native systems," *Journal of Software: Evolution and Process*, vol. 35, no. 10, p. e2413, 2023.

[141] G. Jeh and J. Widom, "Scaling personalized web search," in *Proceedings of the 12th international conference on World Wide Web*, 2003, pp. 271–279.

[142] L. Huang and T. Zhu, "tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces," in *Proceedings of the ACM Symposium*

on Cloud Computing, 2021, pp. 76–91.

[143] Y. Zhang, R. Isaacs, Y. Yue, J. Yang, L. Zhang, and Y. Vigfusson, "Latenseer: Causal modeling of end-to-end latency distributions by harnessing distributed tracing," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, 2023, pp. 502–519.

[144] P.-F. Denys, Q. Fournier, and M. R. Dagenais, "Distributed computation of the critical path from execution traces," *Software: Practice and Experience*, vol. 53, no. 8, pp. 1722–1737, 2023.

[145] P. Janardhana Rao, K. Nageswara Rao, S. Gokuruboyina, and K. N. Neeraja, "An efficient methodology for identifying the similarity between languages with levenshtein distance," in *International Conference on Communications and Cyber Physical Engineering 2018*. Springer, 2024, pp. 161–174.

[146] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 469–478.

[147] P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen, "Cloudranger: Root cause identification for cloud native systems," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 492–502.

[148] C. Zhang, X. Peng, C. Sha, K. Zhang, Z. Fu, X. Wu, Q. Lin, and D. Zhang, "Deeptralog: Trace-log combined microservice anomaly detection through graph-based deep learning," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 623–634.

[149] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 2016, pp. 207–218.

[150] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective software fault localization using an rbf neural network," *IEEE Transactions on Reliability*, vol. 61, no. 1, pp. 149–169, 2011.

[151] D.-Q. Zou, H. Qin, and H. Jin, "Uilog: Improving log-based fault diagnosis by log analysis," *Journal of computer science and technology*, vol. 31, no. 5, pp. 1038–1052, 2016.

[152] S. Lu, B. Rao, X. Wei, B. Tak, L. Wang, and L. Wang, "Log-based abnormal task detection and root cause analysis for spark," in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 389–396.

[153] N. Gurumdimma, A. Jhumka, M. Liakata, E. Chuah, and J. Browne, "Crude: Combining resource usage data and error logs for accurate error detection in large-scale distributed systems," in *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2016, pp. 51–60.

[154] S. Petrescu, F. Den Hengst, A. Uta, and J. S. Rellermeyer, "Log parsing evaluation in the era of modern software systems," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 379–390.

[155] E. Karlsen, X. Luo, N. Zincir-Heywood, and M. Heywood, "Benchmarking large language models for log analysis, security, and interpretation," *Journal of Network and Systems Management*, vol. 32, no. 3, p. 59, 2024.

[156] Z. Dai, "Transformer-xl: Attentive language models beyond a fixed-length context," *arXiv preprint arXiv:1901.02860*, 2019.

[157] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

[158] M. Poli, S. Massaroli, E. Nguyen, D. Y. Fu, T. Dao, S. Baccus, Y. Bengio, S. Ermon, and C. Ré, "Hyena hierarchy: Towards larger convolutional language models," in *International Conference on Machine Learning*. PMLR, 2023, pp. 28 043–28 078.

[159] O. Rubin and J. Berant, "Long-range language modeling with self-retrieval," *arXiv preprint arXiv:2306.13421*, 2023.

[160] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.

[161] V. Patil, P. Hase, and M. Bansal, "Can sensitive information be deleted from llms? objectives for defending against extraction attacks," *arXiv preprint arXiv:2309.17410*, 2023.

[162] J. G. Wang, J. Wang, M. Li, and S. Neel, "Pandora's white-box: Increased training data leakage in open llms," *arXiv preprint arXiv:2402.17012*, 2024.

[163] B. Yan, K. Li, M. Xu, Y. Dong, Y. Zhang, Z. Ren, and X. Cheng, "On protecting the data privacy of large language models (llms): A survey," *arXiv preprint arXiv:2403.05156*, 2024.

[164] B. Gregg, "The flame graph: This visualization of software execution is a new necessity for performance profiling and debugging." *Queue*, vol. 14, no. 2, p. 91–110, Mar. 2016. [Online]. Available: https://doi.org/10.1145/2927299.2927301

[165] V.-H. Le and H. Zhang, "Log parsing: How far can chatgpt go?" in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1699–1704.

[166] K. Guu, K. Lee, Z. Tung, P. Pasupat, and M. Chang, "Retrieval augmented language model pre-training," in *International conference on machine learning*. PMLR, 2020, pp. 3929–3938.

[167] 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," *arXiv preprint arXiv:2311.05232*, 2023.

[168] Y. Lyu, Z. Li, S. Niu, F. Xiong, B. Tang, W. Wang, H. Wu, H. Liu, T. Xu, and E. Chen, "Crud-rag: A comprehensive chinese benchmark for retrieval-augmented generation of large language models," *ACM Transactions on Information Systems*, 2024.

[169] R. Nogueira and K. Cho, "Passage re-ranking with bert," *arXiv preprint arXiv:1901.04085*, 2019.

[170] M. Besta, A. Kubicek, R. Niggli, R. Gerstenberger, L. Weitzendorf, M. Chi, P. Iff, J. Gajda, P. Nyczyk, J. Müller *et al.*, "Multi-head rag: Solving multi-aspect problems with llms," *arXiv preprint arXiv:2406.05085*, 2024.

[171] S. K. Dam, C. S. Hong, Y. Qiao, and C. Zhang, "A complete survey on llm-based ai chatbots," *arXiv preprint arXiv:2406.16937*, 2024.

[172] C. Macdonald, D. Adeloye, A. Sheikh, and I. Rudan, "Can chatgpt draft a research article? an example of population-level vaccine effectiveness analysis," *Journal of global health*, vol. 13, 2023.

[173] R. Aghili and other authors, "An empirical study of sensitive information in logs," 2024. [Online]. Available: https://arxiv.org/abs/2409.11313

[174] S. L. Devi and R. R. Pattabhi, "Intent detection and zero-shot intent classification for chatbots," in *Proceedings of the 20th International Conference on Natural Language Processing (ICON)*, 2023, pp. 636–640.

[175] W. Jiao, W. Wang, J.-t. Huang, X. Wang, S. Shi, and Z. Tu, "Is chatgpt a good translator? yes with gpt-4 as the engine," *arXiv preprint arXiv:2301.08745*, 2023.