Candidate: Alcides Mignoso e Silva

Contact: *alcidesmig@gmail.com*

Non-Intrusive Real-time Dependency Mapping for Microservices in Kubernetes

São Carlos - SP

May 2025

Mapeamento de Dependências Não Intrusivo em Tempo Real para Microsserviços no Kubernetes

Exame de Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Ciência da Computação da *Universidade Federal de São Carlos*, como requisito necessário à obtenção do grau de Mestre em Ciências no Domínio da Ciência da Computação.

Área de Concentração: Sistemas Distribuídos

Discente: Alcides Mignoso e Silva

Orientador: Prof. Dr. Hélio Crestana Guardia

São Carlos - SP, Maio de 2025

Acknowledgments

This work was only possible thanks to the support of several people who, in different ways, contributed to both my academic and personal journey.

I would like to thank my advisor, Prof. Dr. Hélio Crestana Guardia, for his trust, patient guidance, and the insightful technical discussions that greatly enriched this research. His unwavering dedication and constant encouragement were essential for the completion of this dissertation.

To my family, for their unconditional love and support at every stage of my path. I am especially thankful to my partner, Vitoria Cristina Peres do Prado, for her continuous support, encouragement, and understanding throughout the development of this work – none of this would have been possible without her and her support in the most difficult moments was what made me get here. To my parents, Edilson Souza e Silva and Luciana Mignoso e Silva, for their lifelong support, their example, and for teaching me to see the world the way I do. Thank you for always believing in me.

To everyone involved in the research and infrastructure projects at UFSCar, thank you for the opportunities to grow and learn in a collaborative and inspiring environment.

To Magalu Cloud, for offering me a rich and dynamic environment where many of the ideas behind this dissertation took shape, and for providing the physical infrastructure used in the experiments.

To the Cilium community and support channels, which were crucial in discussing approaches and clarifying doubts, especially Chance Zibolski, who kindly and thoroughly enriched technical discussions on several occasions.

Finally, to all the people who, directly or indirectly, helped make this work possible – my sincere thanks.

"The only normal people are the ones you don't know very well."

(Alfred Adler)

Abstract

The ever-evolving landscape of microservice architecture demands innovative solutions for improved performance and environment management, especially under varying request loads. While strategies developed for managing monolithic systems have been effective in the past, they may not be effective in dealing with the challenges of the microservice-based applications, which often present a high level of interdependency between services.

Considering the potential benefits of knowing the relationships among microservices and understanding their behaviors, this work introduces a modular and adaptive tool that maps the microservice dependencies within systems and use the data to provide insightful information about the services' communications patterns.

The central hypothesis this study posits is that by using existing network tools, such as Cilium, it is possible to automatically map service dependencies and communication ratios in a microservices environment and use the connections information to generate dependency data that can serve as input for application management.

Given its versatility and large adoption, Kubernetes is the chosen development platform, enhancing the scope and usability of the results obtained with this research. Cilium and its eBPF-based technology are used for collecting the information needed for automatically creating the envisioned microservices communication maps, and authoring applications are introduced to transform and process the collected information.

The results of this research may benefit practitioners seeking to optimize the architecture of their microservice-based applications, and also contribute to the broader field of application development by emphasizing the importance of considering services dependencies and characteristics in services management. Thus, the findings shall inspire future research endeavors to explore innovative approaches in addressing the unique challenges posed by microservices architecture by utilizing the intrinsic communication information between services.

Keywords: Microservices Architecture, Service Dependencies, Non-Intrusive Automatic Mapping.

Resumo

O cenário em constante evolução da arquitetura de microsserviços requer soluções inovadoras para aprimorar o desempenho e o gerenciamento de ambientes, especialmente sob cargas de tráfego variáveis. Embora as estratégias desenvolvidas para o gerenciamento de sistemas monolíticos tenham sido eficazes no passado, elas podem não ser eficazes para lidar com os desafios das aplicações baseadas em microsserviços, que frequentemente apresentam um alto nível de interdependência entre serviços.

Considerando os potenciais benefícios de conhecer as relações entre microsserviços e compreender seus comportamentos, este trabalho apresenta uma ferramenta modular e adaptável que mapeia as dependências de microsserviços dentro de sistemas e utiliza os dados para fornecer informações valiosas sobre os padrões de comunicação dos serviços.

A hipótese central deste estudo é que, utilizando ferramentas de rede existentes, como o Cilium, é possível mapear automaticamente as dependências de serviços e as taxas de comunicação entre eles e utilizá-las para gerar dados de dependência que podem servir de entrada para o gerenciamento do ambiente.

Dada sua versatilidade e ampla adoção, o Kubernetes é a plataforma de desenvolvimento escolhida, aumentando o escopo e a usabilidade dos resultados obtidos com esta pesquisa. O Cilium e sua tecnologia baseada em eBPF são utilizadas para coletar as informações necessárias para a criação automática dos mapas de comunicação de microsserviços, e aplicativos de autoria são introduzidos para transformar e processar as informações coletadas.

Os resultados desta pesquisa podem beneficiar profissionais que buscam otimizar a arquitetura de seus sistemas baseados em microsserviços e também contribuir para o campo mais amplo do desenvolvimento de aplicações, enfatizando a importância de considerar dependências e características de serviços no gerenciamento de ambientes. Assim, os resultados devem inspirar pesquisas futuras para explorar abordagens inovadoras para lidar com os desafios únicos impostos pela arquitetura de microsserviços através da utilização das informações intrínsecas de comunicação entre os serviços.

Palavras-chave: Arquitetura de Microsserviços, Dependências de Serviços, Mapeamento Automático Não Intrusivo.

List of Figures

Contents

1 Introduction

Over the past few decades, the development of software systems has gone through important transformations, largely driven by the need for scalability, flexibility, and faster delivery cycles. In this context, the microservices architecture has emerged as an alternative to the traditional monolithic model, decomposing applications into small, autonomous, and specialized components. However, this shift also introduced new challenges, especially regarding observability, resource management, and services communication.

Understanding how services communicate with each other within a system is essential in a distributed system environment, helping from troubleshooting and debugging to performance tuning, security analysis, and autoscaling. The decentralized nature of the microservices architecture makes it hard to maintain an up-to-date and accurate view of service-to-service communication patterns. While existing observability tools – such as service meshes and distributed tracing platforms – attempt to address these challenges, many require manual instrumentation, the use of sidecars or intrusive methods, and end up not making the best use of what is already used to connect services: the network.

This research proposes a low-overhead and non-intrusive method to automatically map dependencies between microservices running in Kubernetes clusters. By using Cilium and Hubble, the method captures real-time network traffic at both Layer 4 and Layer 7 to infer service interactions. Based on this data, the system constructs dependency graphs that model how applications communicate under different workloads, enabling further analysis and insight extraction.

The proposed approach was validated using controlled experiments on open-source microservices systems. The findings demonstrate that the automatically generated graphs reflect actual system behavior and offer useful data for service management. This work contributes to the field of microservices observability by offering a practical solution to dependency mapping, with potential applications in system visualization, anomaly detection, and architecture optimization.

1.1 Motivation and Objectives

As microservices-based systems grow in scale and complexity, managing resources efficiently and understanding the behavior of distributed applications becomes increasingly difficult. Since services often rely on each other to complete their tasks due to their distributed nature, the performance and availability of a system consequently depend on hidden or indirect dependencies. These dependencies, when not clearly identified, can lead to incorrect assumptions and complicate management scenarios.

The main goal of this research is to improve the visibility of microservices communication by using network-embedded tools. By leveraging existing technologies already present in the network layer – specifically Cilium and Hubble – it is possible to extract telemetry data and transform them into visual representations of the system's dependency structure.

The key objective is to develop and validate a model that uses Cilium to automatically detect service dependencies inside a Kubernetes environment, map these relationships into structured metrics and demonstrate how these data can be used for better system understanding and decision-making.

1.2 Contributions

The main contributions of this work are:

- A non-intrusive methodology for real-time dependency mapping in Kubernetes-based microservices, requiring no code instrumentation or application modification.
- An implementation of a processing layer that transforms Hubble's telemetry into dependency graphs, with support for both direct and indirect dependency detection.
- A definition and validation of dependency degree metrics that can be used to quantify service importance or centrality in the system.
- A set of experiments and analysis that demonstrate how these metrics can explain real performance issues and help identify key services in distributed environments.
- A foundation for future research on topology-aware autoscaling, anomaly detection, and resource allocation.

1.3 Organization

This research document is organized as follows:

- **Chapter 2** presents a review of the existing literature on monolithic and microservices architectures, including challenges in resource management, observability, and dependency analysis.
- **Chapter 3** describes the research context, objectives, and methodology, detailing how the proposed system captures and processes network data to build service dependency graphs.
- **Chapter 4** discusses the experimental setup, data collection process, and results obtained through tests on real microservices benchmarks.
- **Chapter 5** concludes the work with a discussion of findings, limitations, and suggestions for future research directions.

2 Contextualization

This section provides a contextualization of the evolution of systems architecture paradigms, outlining the transition from monolithic systems to microservice-based architectures. It highlights the motivations behind this shift, the advantages and challenges inherent to each model, and the implications for modern system design, deployment, and maintenance.

2.1 Monoliths

Until the consolidation of the service-based architecture model, large systems for years were mainly composed of a single large component or of a small limited number of substantial components, commonly referred to as "Monoliths" – which means "too large", according to the Cambridge dictionary[1]. Monolithic applications were responsible for encompassing all business logic and services of their context and were considered for years the most common way to develop and deploy applications.

To exemplify the coverage of a monolithic application, it is possible to think of an entire e-commerce system built on a single code base and deployed as a single application. That could include: the recommendation system, ordering, search, discounts, payments, ads, and all other parts of an e-commerce in one unique system. An example of such a system was built in the DeathStarBench Benchmark Suite [28], and used for testing and comparison purposes. In this architectural pattern, even if the source code is logically modular, it is packaged and deployed as a single application.

2.1.1 Monolithic applications

To improve understanding of the monolithic architecture, Figure 1 illustrates the number of components involved in the deployment of both monolithic and microservice-based applications. The latter will be elucidated in subsequent sections. Notably, the left part of the image delineates the inherent level of coupling associated with the monolithic architecture.

In a monolithic application, the replication of a large singular component across multiple

[1]Definition of "monolith" from the Cambridge Dictionary: https://dictionary.cambridge.org/us/dictionary/english/monolith

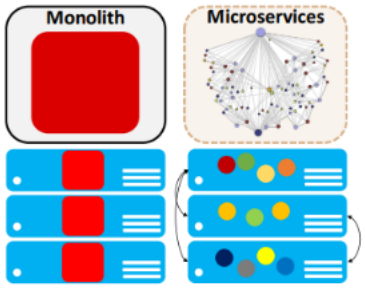


Figure 1: Differences between monolithic and microservices architecture and deployment (Figure 1 of DeathStarBench[28]).

machines is used to ensure enhanced availability. This requires complete system replication, as all services are bundled as a unified entity.

An observation can be made regarding the necessity to eventually modify the system, by adding new features, fixing bugs, etc. The image can exemplify the scenario in which one small modification can cause the replacement of the entire system, including all deployments and locations. Simultaneously, one can also recognize the simplicity in comprehending the components of the system and its organizational structure.

In the alternative approach, it is noteworthy to highlight the directional arrows that denote communication between services. Due to the system's division into smaller parts, the applications, it is imperative for them to interact and collaborate in executing tasks. In contrast, within a monolithic architecture, all functions can be encompassed within a single entity, thereby mitigating numerous challenges inherent in distributed systems.

The primary objective of the work presented here is not to extensively delve into the intricacies of the monolithic architecture. However, it is crucial to understand it to grasp the historical backdrop of microservices. Hence, acquiring knowledge about the fundamental aspects of this architecture is of value and significance in comprehending the circumstances that gave rise to microservices. The subsequent list outlines the principal discussions and characteristics found in the literature regarding the monolithic architecture:

- Simplicity: having all the context and code into one unique application facilitates the process of testing, deploying, debugging, and monitoring [10][52];

- Synchronization: as all tasks are done within one unique system, many concerns about

applications communicating and the use of intermediary systems are solved, without communicational overhead [28][37]. The existence of one unique database/datastore also facilitates data synchronization and management [10];

- Startup time: as one unique application loads the entire system, the applications take longer to be loaded [10][52];
- Development: as the code grows, it becomes more complex and difficult to maintain, leading to losses in productivity [10][37][62][54];
- Reliability: errors are common, and even the finest processes can not guarantee the absence of errors. When running monoliths, even localized errors can lead to the failure of the entire system;
- Scalability: since the applications are monolithic, in heavy load scenarios it is necessary to scale the entire application instead of scaling only the part which is saturated [10][37][78][8]
- Performance: discussions lead to the understanding that the performance of monolithic systems can fluctuate based on the workload style [28][69], but in general cases, they have higher performance when compared with microservices, as discussed on [28][10][62][8][2].

2.2 Microservices

2.2.1 Definition

Although monolithic architecture has several positive points, in recent years the architecture of microservices has dominated the scene. While in monoliths entire complex systems are grouped into a single application, in the microservice-based architecture the proposal is to split the system into individual modules, based on their responsibilities.

The microservice architecture brings a highly decentralized approach based on independent services and inspired by service-oriented computing [36][67]. Chris Richardson defines

microservices as *"an architectural style that structures an application as a collection of services that are: Independently deployable; Loosely coupled; Organized around business capabilities; Owned by a small team; Highly maintainable and testable"* [60], while Amazon defines it as *"an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs [...] owned by small, self-contained teams"* [64]. More formal and simplified definitions can be found in [77]: *"Microservices are autonomous components that isolate fine-grained business capabilities"*, and [24]: *"A microservice architecture is a distributed application where all its modules are microservices"*.

Despite variations in the definitions of this architecture, all adhere to the initial definition put forth by James Lewis and Martin Fowler in 2014 [26]: *"it is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery"*.

The microservices architecture has numerous advantages and disadvantages. Faster delivery, improved scalability, and greater autonomy are also cited by Lewis [26] and "Microservices: The Journey So Far and Challenges Ahead" [36] as the most important benefits associated with microservices. On the other hand, both works also delve into the inherent challenges associated with this architecture, which include definitions beyond service modularization and refactoring, lack of agreement on the right size of microservices, team coordination, resource monitoring and management, and problems inherent in distributed systems such as fragility, partial outages, and communication failures.

A survey on microservices architecture conducted in "Microservices in Practice: A Survey Study" [77] identified additional advantages for microservice-based applications, including the possibility of deploying services independently, the individual and on-demand scaling, and the benefits involved in maintaining systems in isolated ways. In the same survey, some challenges are also presented: the distributed data management inherent in distributed systems, the growth of the complexity in integration tests, the difficulty of identifying service faults, and the cost of the commonly used Remote Procedure Calls (RPC) - the latter being

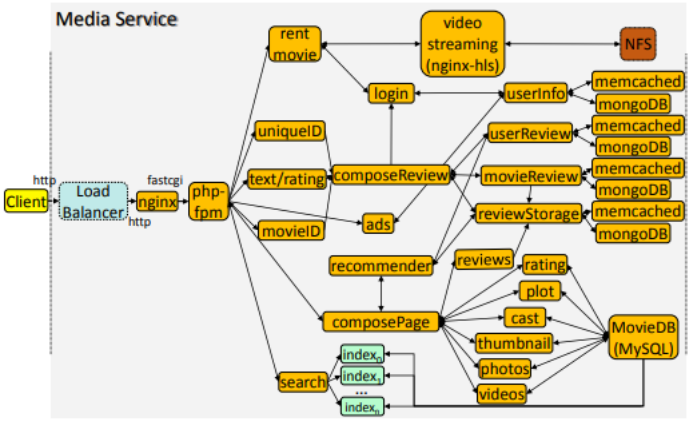


Figure 2: Applications of a Media Service system for reviewing, renting, and streaming movies (Figure 10 of DeathStarBench[28]).

questioned by interviewed practitioners.

It is important to note that even if the concept of a microservice-based architecture is relatively recent, it is possible to find several researches on the subject, as it has become a very popular topic. However, many gaps and challenges are still being discovered.

2.2.2 Example

In order to provide a more illustrative depiction of a system composition using the microservices architecture, Figure 2 showcases a Media Service system designed to review, rent and stream movies. This system was specifically developed for the test suites of "DeathStarBench" [28], an Open-Source Benchmark Suite for studying microservices.

For the purpose of this study, we shall refer to a complete system as a "system", while its constituent elements (microservices) shall be referred to as "applications".

In the image presented in Figure 2, it is observable that each yellow box fulfills a unique functionality, and their collective integration forms the system. In a monolithic approach, otherwise, most yellow boxes would typically be incorporated within a single system with a unique implementation (as exemplified in Figure 1) – in the current example, each box possesses its own implementation, code base, and particularities.

2.2.3 Critical points

The microservices architecture was developed as an alternative to the monolithic architecture, but it also originated other explicit and implicit issues, which are still being discovered and studied. This section discusses how microservices problems correlate with the problems considered the critical points of general systems: availability, reliability, communication, maintainability, testability, and security [24].

Availability is also a key factor for any system, and the microservice architecture brings some advantages regarding the theme. As services are independent, [24] argues that the availability of whole systems can be measured by the availability of their individual systems, since failure of a single service may not compromise the entire system. However, some authors say that with the increase in the number of services, the fault-proneness at the integration level grows, leading to complex problems managing the various services available. Martin Fowler [26] contours this explaining that since any service call is liable to fail, the applications need to be designed to be able to detect and tolerate the failure of services, implementing circuit breakers and other fault tolerance strategies – which is harder to do in monolithic systems.

To illustrate these aspects of availability, we can revisit the aforementioned Figure 2, where we can observe a component named "userReview", which encompasses the system responsible for movie reviews by users. In the event of a failure of this application, if the microservice architecture is properly implemented, users would still be able to utilize other functionalities of the system, such as video streaming. In a monolithic architecture, if the user review part crashes, it would likely result in the entire system becoming inoperative and unusable. Conversely, certain components of the system have critical significance. In the event of an individual failure of the login system, no users would be able to access the system. For example, if the "phpfpm" component fails, it would result in a complete system outage, as it resides at the core of the call chain.

Therefore, with dozens of services that make up a single system, the reliability of the system within the microservices architecture is not trivial and requires particular attention [24], due to the inherent dependency between services. To achieve it, some techniques can be put into practice, as seen in [33], which used automated quality assurance pipelines with

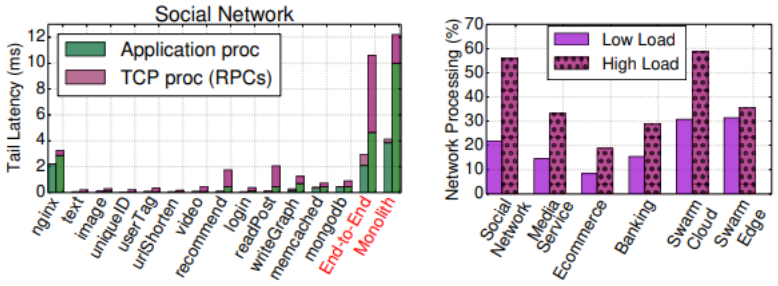


Figure 3: Time spent in the application and within network processing for microservices in Social Network, and comparing with other services from DeathStarBench (Figure 15[28]).

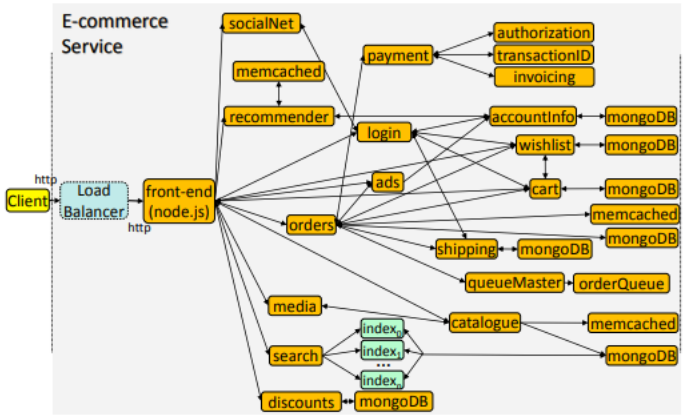


Figure 4: Applications of an E-commerce Service system developed in DeathStarBench work (Figure 6 of [28]).

continuous integration and deployment. In their pipelines, automated building, execution of units, integration, and system tests allowed them to achieve macrosystem reliability while increasing their deployment numbers from 40 to 500 deployments per week in two years.

Communication is definitely the Achilles heel of microservices, and [28] proves it in practice through experiments in a real environment built for testing. Microservices communicate over networks, and network latency is orders of magnitude higher than the time for in-memory calls (commonly used in monoliths), opening doors to a range of communication, synchronization, latency, and performance problems [24]. In their paper, DeathStarBench's authors discuss the *computation:communication* ratio present in the architecture, and found that for some high-load scenarios more than fifty percent of the latency time is due to network processing time, as can be seen in Figure 3.

In communications, the impact of the network becomes evident due to the inherent interdependence among applications. In the scenario shown in Figure 3, an ideal network condition was assumed. However, network fluctuations or situations in which services (applications) communicate across physically distant locations can introduce challenges throughout the entire call chain. To illustrate that, we can consider the microservice architecture of an e-commerce shown in Figure 4. If the "orders" system is hosted in São Paulo (Brazil), and the remaining applications are located somewhere in the Europe, for example, each call to the "orders" system would currently incur a minimum latency of 200 ms for each system. In case a particular action necessitates two separate queries to this "orders" system, the response time would take at least 400 ms. This poses a significant issue, as studies suggest that even a 100-millisecond delay in average page load times can result in a 2.4-7.1% decrease in conversion rates on e-commerce [9][1].

However, the challenges of communication in the microservice architecture extend even further. When considering microservice architectures with a multitude of applications, as is often the case with large companies, and often relying or depending on external systems, it is fair to assert that the network and latency implications remain largely unfamiliar to the broader community. Considering a scenario where two applications communicate in a way that each request to application "A" triggers a request to application "B", if application "B" becomes saturated and exhibits an average response time of 250 ms, a single synchronous thread from application "A" would be capable of handling a maximum of 4 requests per second due to the existing dependency. Later, we will show that this can potentially create a bottleneck in the request queue of application "A", leading to a backpressure scenario that may trigger the scaling of application "A" and further exacerbate the saturation of application "B".

Maintainability is also a pivotal aspect in the context of a microservice-based architecture, entailing a set of inherent trade-offs. In this context, the cost of modifying one service is lower than in other architectures, but it is necessary to manage the code deployment for a greater number of applications. In order to achieve maintainability and reliability, it is necessary to instrument the entire environment (including the network) and each system with tools for traceability and visibility, such as dashboards and alerts. While with monoliths, this

only needs to be done once, in the microservice architecture this work is distributed and more complex. Even so, software development itself benefits from this architecture, since the division of responsibilities between systems allows different teams to control different systems and run software development lifecycle activities in parallel.

Microservice architecture provides some benefits and challenges for testability. Given that each component can be tested separately, it is easier to implement tests for each isolated system, simulating or mocking the other parts of the systems, which is hard to do within monoliths. The challenge starts with integration tests, where the connection between components can lead to traceability issues, cascading errors, and implicit or hidden bottlenecks. In this scenario, error traceability is critical, and the community has developed many tools to help with it, such as the OpenTelemetry collection [75].

Security in monoliths can often be addressed at a single point in the system, basically its entry point. However, in microservices, the attack surface expands, with each system being a potential entry point for malicious requests. Proper authentication and authorization mechanisms must be implemented to ensure that only authorized services can access sensitive data and functionality, addressing service-to-service security communications issues and often depending on systems of authority and identity. It is imperative for systems to establish their identities during calls and ensure that the calls are made to authenticated systems. Strategies such as mTLS (mutual Transport Layer Security) or the utilization of OAuth [17] specific flows - such as the "client_credentials" one - between systems become indispensable. Therefore, the community has developed numerous tools to facilitate the implementation of these standards. Open-source systems and service meshes, such as Istio [72], can be used to abstract the resolution of these challenges, providing a good approach to address these issues.

In conclusion, the adoption of microservice architecture has been fueled by its ability to tackle major challenges that exist in other architectural approaches. By enabling independent services and enhancing availability, reliability, maintainability, testability, and security, microservices offer valuable benefits to the development of complex software systems. However, it is crucial to recognize that this approach introduces its own complexities and potential inefficiencies, particularly in terms of communication and coordination among services.

Each system must be carefully evaluated, taking into account its specific requirements, and considering the trade-offs associated with different architectural choices. Striking the right balance and making informed decisions will ultimately lead to successful implementations of the microservice architecture, harnessing its advantages, while mitigating its challenges.

2.2.4 Current state

This section presents a bibliographic review of the current state of microservices research. Recognizing the relative newness of the subject, it highlights key scholarly contributions and groundbreaking studies that have shaped studies around microservices. Although the literature is still in development, some notable works have contributed significantly to our understanding of this architectural style. This section emphasizes some of the opportunities and challenges that lie ahead, emphasizing the need to continue exploration and innovation in the microservices domain.

The authors of [11] assert that microservices research is still in a formative stage, with the development of a systematic mapping study to assess the state of the published literature regarding methods of microservices analysis and architecture. The article focuses on the analytical part of microservices and categorizes the approaches and techniques used in microservice analysis. The paper then identifies the existing problems and challenges in the scenario and discusses the relationship inherent in existing and advanced architectures. The conclusion states that, although the influence and impact of this architecture have grown a lot since its conception, there is space for improvement on many fronts, such as in migration strategies (properly dividing systems into microservices), microservice analysis (open to innovations and new methodologies) and mostly about robust architecture reconstruction in order to better face system understanding, evolution, and degradation. The researchers cite the division of complex systems into microservices and the complexity of systems resulting from this as special challenges and show that interest in the analysis of this architectural style is growing, which is particularly good for the future. The paper also discusses issues and solutions regarding the design of centralized architecture and discusses architectural degradation.

As mentioned previously, [36] illustrates many existing challenges associated with the

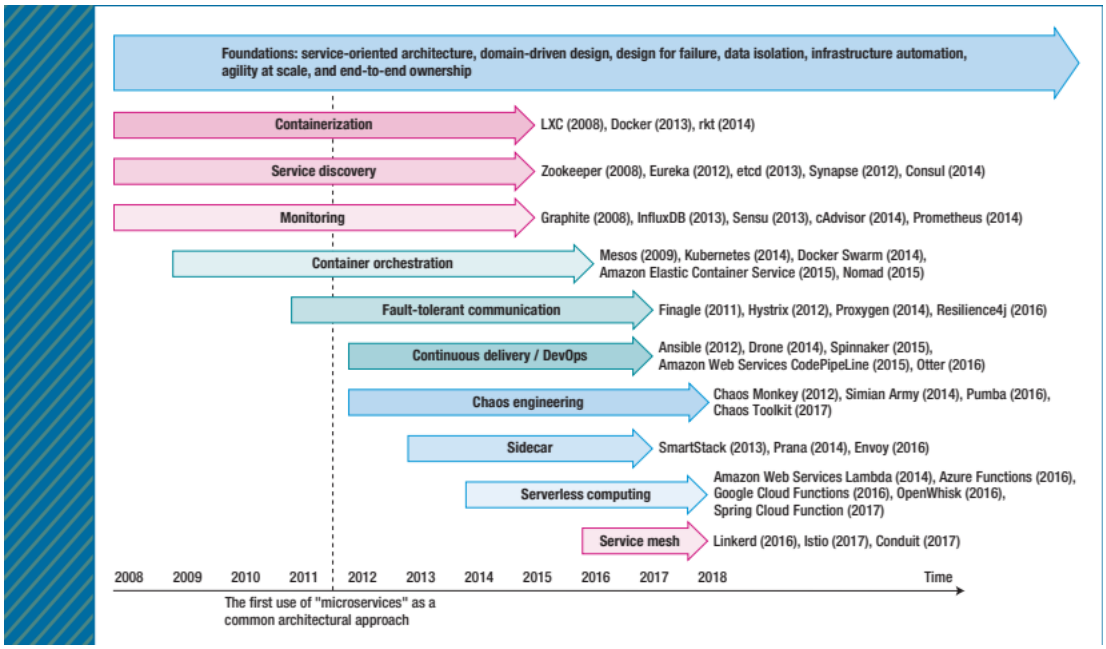


Figure 5: Microservice waves of technologies timeline (Figure 1 of [36]).

use of microservices and also presents a historical perspective on the topic. The article explains that although the term microservices emerged in 2011, industry experts were already experimenting with the idea before. Terms such as "fine-grained SOA", "SOA done right" and "loosely coupled service-oriented architecture with bounded contexts" were cited, demonstrating that the industry was not satisfied with the SOA approach and clarifying the inspiration of microservices on top of it. It also mentions other influencing factors, such as domain-driven design (DDD), design for failure, data isolation, infrastructure automation, agility at scale, cross-function teams, and end-to-end product ownership [26]. With regard to technological aspects, the article reflects on the influence of the new generation of software development, deployment, and management on microservice architecture, which allowed the creation of even more advanced tools. Figure 5 presents the waves of technologies that influenced the development of the microservice architecture ecosystem, and the article discusses how tools from the first wave (LXC, Docker) to the last (Service Mesh: Linkerd, Istio) influenced the design of microservices to achieve what is the state-of-the-art at the present time.

"Microservices: Yesterday, Today, and Tomorrow" [24] was published in 2017 and provided formal definitions for the microservice ecosystem and goes through the past, current

scenario, and future prospects for it. From the past, called "Yesterday" by the authors, the study discusses the problems associated with large-scale software development from the 1960s to the 2000s, explaining architectural patterns that have emerged since the beginning of software development and landing in Service-Oriented Computing (SOC) and SOA. Based on SOA concepts, the "Today" section discusses the microservice trend in current software architecture, differing from SOA by size, bounded context, independence, flexibility, modularity, and evolution. The authors also state that the microservice architecture gained popularity recently and can be considered in its infancy due to the lack of consensus in its definitions. The "Tomorrow" section suggests that the exploration of microservices is still in its early phases, pointing out new challenges that arise due to the inherent difficulties of programming distributed systems, such as issues related to dependability, trust, and security. For the proposal presented in the next sections, an intriguing question raised by the authors was: "how can we manage changes to a service that may have side effects on other services that it communicates with? ". The conclusion argues about the "evolutionary" word instead of the "revolutionary" one for describing microservice architecture, with a significant potential for further evolution and arguing about notable gaps and deficiencies in the existing literature, which can be seen as indicators of the novelty and emerging nature of the subject.

Given the recent and rapid emergence of this architectural paradigm, the microservices architecture paradigm is still subject to several misconfigurations and oversights. Empirical evidence collected by [68], based on interviews with 72 experienced developers, has identified several detrimental practices, which have been classified into 11 distinct "bad smells". The most cited bad smells were "Wrong Cuts" (systems split into microservices in the wrong way), "Hard-Coded Endpoints", and "Shared Persistency" (multiple services accessing the same database), but other 9 smells and 5 lessons were described. In a separate study, [18] conducted interviews with 25 practitioners to identify sixteen distinct Architectural Technical Debts (ATD) issues, along with their negative impacts and potential solutions. Figure 6 presents a mapping of twelve of these issues. Notably, there was some agreement between the findings of these two surveys, such as issues related to shared databases and API design. It is worth bringing up that many of these issues can be solved without significant difficulty and, as highlighted in lesson 2 of the first survey, the role of the software architect has

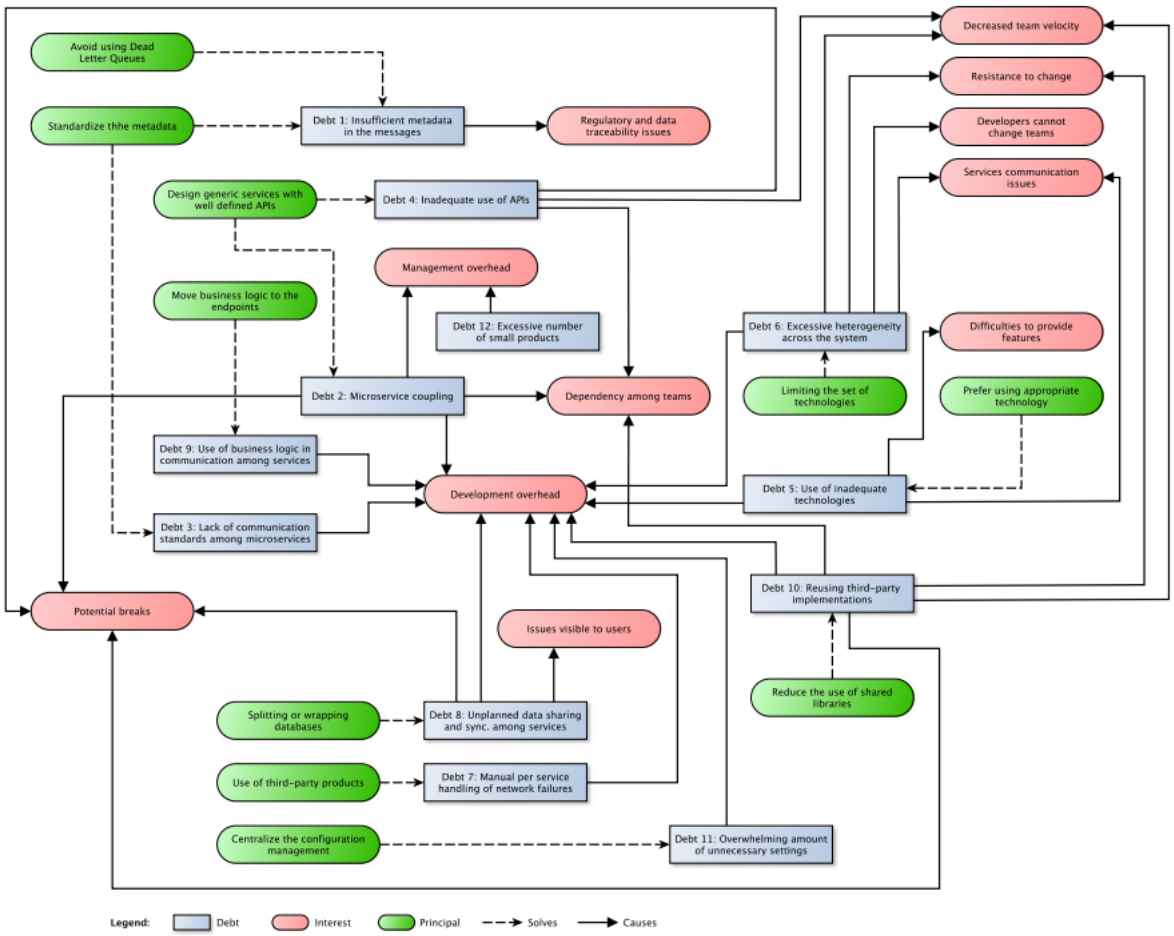
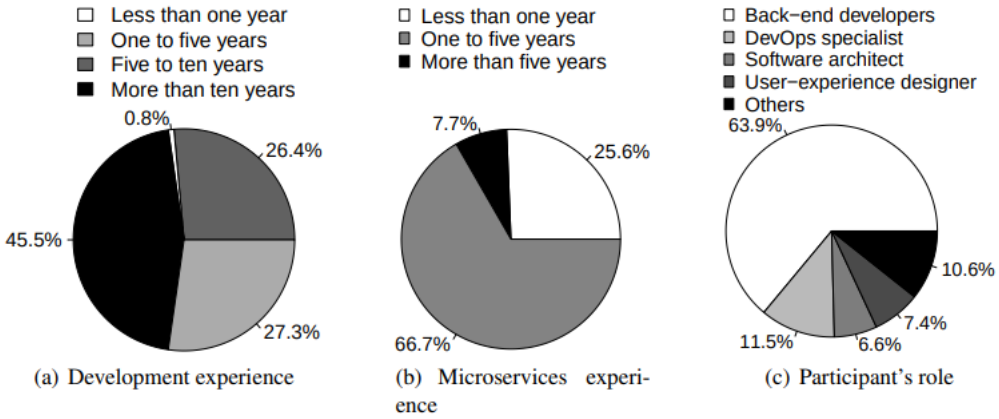


Figure 6: Common interests and principals among the debts in microservices architecture context (Figure 13 of [18]).

become increasingly vital, as system-level decisions must be made with a comprehensive understanding of microservices architecture in order to mitigate these common issues.

Another mapping study and survey was conducted by [77]. The mapping study of the article, which has been previously presented, addressed aspects regarding the advantages and challenges of microservices and was used in a survey comprised of a set of 14 questions. The survey was completed by 122 professionals who actively participate in microservices development. Figure 7 illustrates the findings, revealing a notable observation: despite a sample size of more than 45% comprising developers with more than 10 years of experience, less than 8% had more than 5 years of experience specifically with microservices. This observation accentuates the notion that microservices are a relatively new and emerging



(a) Development experience (b) Microservices experience (c) Participant's role

Figure 7: Background and experiences of the "Microservices in Practice: A Survey Study" survey's participants (Figure 1 of [77]).

field.

Furthermore, the article highlights the programming languages commonly used in microservices development, with Java and JavaScript ranking as the top two choices among developers. Additionally, the preferred communication protocol among these professionals is REST over HTTP. Also, it is important to acknowledge that the article in question was published in 2018 and, since then, the microservice landscape has likely evolved significantly. Notable changes may include the emergence of new tools within the ecosystem, such as the Golang and Python programming languages and specialized microservices development tools. Nonetheless, it is intriguing to observe that many of the challenges identified by the mapping and voted on by the participants, as depicted in Figure 8, remain pertinent today, underscoring their fundamental nature within the microservices architecture.

2.2.5 Characterizations and implications

Although microservices remain a topic with multiple definitions and create some divisions among service-oriented approaches, the existing literature has already identified interesting characteristics of this architecture.

The DeathStarBench study [28] provides valuable insights into the architectural traits of microservices and their implications in various environments. The researchers found that a significant portion of CPU cycles is often expended in the processor front-end, primarily due

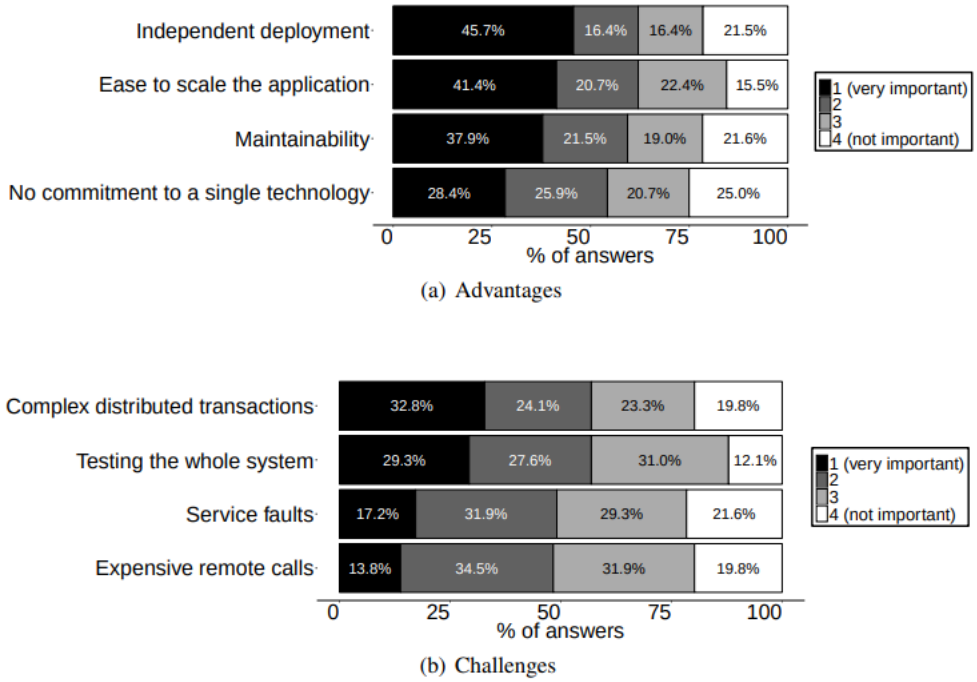


Figure 8: Advantages and challenges votes of "Microservices in Practice: A Survey Study" survey's participants (Figure 2 of [77]).

to factors such as lengthy memory accesses and instruction cache (i-cache) misses. However, microservices exhibit lower i-cache pressure compared to cloud services due to their inherent simplicity, including smaller codebases and specific responsibilities. Consequently, microservices demonstrate improved i-cache locality. Furthermore, the study highlights the increased sensitivity of microservices to poor single-thread performance compared to traditional cloud applications, particularly for interactive services that are also affected by frequency scaling.

At the operating system and networking level, the authors of the DeathStarBench article emphasize that microservice applications, due to their development nature, spend a significant portion of their execution time in library instructions and kernel cycles. This finding relates directly to the network dependency in microservices discussed earlier, and the authors also provide insights related to the computation-to-communication ratio of this architecture. For common microservices, RPC processing accounts for 5-75% of the execution time and contributes to 18% of the end-to-end tail latency in their microservices-based Social Network under low-load conditions. In high-load scenarios, network processing causes the end-to-end

tail latency to increase by 3.2 times due to the accumulation of long queues in the Network Interface Controllers (NICs). Importantly, the network's substantial impact on microservices performance occurs regardless of the communication architecture employed (e.g. RPC, and HTTP) and is inherently linked to the existence of numerous applications used for service composition.

Another aspect worth noting is the latency composition of services within the microservices architecture. Under low load, typical microservice applications experience latency dominance at the front-end level (e.g. nginx in the case of DeathStarBench). However, during high-load situations, latency is predominantly influenced by back-end databases and their management systems. Consequently, the overall conclusion is that microservices encounter shifting bottlenecks with varying loads, making their management and scaling even more challenging.

In large-scale scenarios, the DeathStarBench study includes an indispensable examination of the performance impact of dependencies among microservices. In one experiment, a load balancing failure in two small microservices out of the total 36 microservices in their Social Network system resulted in a waterfall problem of latency across tiers. This cascade effect caused downstream services and the entire system to reach saturation levels, which could only be quickly resolved by implementing rate-limiting techniques. This empirical evidence demonstrates that the dependency between services can lead to cascading QoS degradations through relatively simple and unusual oscillations. Furthermore, the study reveals that even a single slow server among a hundred servers can lead to QoS problems. It concludes that "the more complex an application's microservices graph, the more impactful slow servers are, as the probability that a service on the critical path will be degraded increases", subsequently contributing to the aforementioned problems.

[50] made a comprehensive analysis of more than 20000 microservices in a 7-day period. They observed that microservices call graphs follow a heavy-tail distribution, with a tree-like structure with few hotspots and a lot of services in critical paths and up to nine classes of topologically different graphs in extreme cases. They stated that, in Alibaba systems, 5% of the critical paths appear in more than 50% of their call graphs. This low number of hotspots for general call graphs indicates that resource management solutions should focus more on

the critical paths of the systems for achieving high efficiency.

In the same context, the authors present three significant observations regarding the progression of microservices applications in the studied environments. Firstly, long-term developed applications exhibit a substantial increase in the number of services, approximately twelve times greater than that of short-term applications. Secondly, these long-term applications manifest a hierarchical structure resembling a tree, with no more than four tiers. Finally, long-term applications demonstrate a higher degree of loose coupling compared to their short-term counterparts. These findings imply that, as time elapses, systems expand regarding to the number of applications, albeit with reduced coupling and fewer tiers. This observation aligns with the intuitive notion that such evolution should address encountered challenges.

The researchers additionally observed a fluctuation pattern in the Microservice Call Rate (MCR), which exhibited periodic changes in behavior across different microservices over the course of several days. This fluctuation can be attributed to the dynamic and intricate call dependencies between microservices. Consequently, it can be inferred that the level of interdependence between applications can vary within a given scenario. For instance, in the case of customers adding more shopping items during the night, the communication frequency between the cart system and the product system would increase. Such variability in behavior further complicates the comprehension of the communication landscape within a system.

Still in [50], the authors also have drawn several intriguing conclusions concerning the performance of microservices. They assert that microservice call rates exhibit a strong correlation with CPU utilization and garbage collection (GC), but not with memory utilization. Notably, memory utilization remains relatively stable at runtime in most containers within the Alibaba microservices environment. This finding suggests that relying solely on memory data for scaling applications is entirely inefficient. Since memory management is more expensive than CPU management and has been less explored, some solutions can lead to wasteful memory issues. To address this concern in Alibaba applications utilizing JVM, the authors recommend managing cluster memory based on heap memory at runtime rather than container memory - due to JVM's behavior of not returning released memory to the container, potentially leading to incorrect inferences regarding memory utilization.

In the same study, the Mean Call Rate (MCR) demonstrates variability, while the response time of microservices remains stable regardless of call rate fluctuations. However, this conclusion warrants careful consideration as the Alibaba environment benefits from abundant system resources. Considering better-optimized and non-abundant environments, response times may vary due to potential interdependencies between systems leading to cascading scaling challenges. Deciding where to place services is crucial, as findings indicate that balancing CPU utilization across different hosts significantly improves response time performance by avoiding resource interferences. Additionally, collocating dependent microservices on the same host yields an average 22% enhancement in response time performance, effectively reducing communication overhead when dependent microservices are spatially distant from each other.

2.2.6 QoS and metrics

In the domain of microservices architecture, Quality of Service (QoS) pertains to the level of performance, reliability, and availability that a system delivers. In this context, certain advantages and challenges arise. The ability of services to be developed, deployed, and managed independently offers the flexibility for distinct QoS configurations and precise oversight. Therefore, the consideration of communication patterns in this setting becomes crucial, as interdependencies between microservices can result in a ripple effect of QoS issues when certain components in the communication pathway encounter difficulties and end up saturating their entire system.

Frequently, Quality of Service (QoS) is quantified using specific metrics, which encompass a collection of data generated by applications to offer insights into its performance, healthiness, efficiency, and overall behavior. These metrics enable the establishment of thresholds and desired benchmarks, allowing continuous monitoring to discern instances of QoS deterioration and to determine appropriate interventions for amelioration. Given the distinctive nature of microservices applications, the identification and utilization of appropriate metrics hold significance. These metrics need to encapsulate the unique attributes inherent to microservices, often diverging from the conventional metrics employed for more generic applications.

Although not centered exclusively on microservices, the article [34] introduces a System of Systems (SoS) methodology to facilitate the management of Quality of Service (QoS) in cloud-based enterprise systems. The authors emphasize that intricate systems are susceptible to vulnerabilities and security compromises across five principal domains: computational performance, cloud reliability, economic objectives, compliance, and information security. They critique prevailing solutions for their limitations in effectively ensuring QoS, particularly when addressing these key domains.

Furthermore, the authors underscore several metrics that hold paramount importance within the realm of cloud computing and performance assessment. These metrics encompass delay, crucial for deciphering optimal application configurations; delay variation, pivotal for pinpointing resource-intensive operations; throughput, indicative of comprehensive system performance; and metrics pertaining to authentication and authorization, valuable for upholding security objectives.

The 2022 edition of the Google "State of DevOps Report" [30] delineated the DevOps practices that underpin efficacious software delivery and operational efficiency, with an intensified emphasis on security considerations. In a noteworthy extension beyond conventional software delivery metrics, the report introduced the "reliability" metric as the "Fifth Key Metric". This addition was designed to encompass availability, latency, performance, and scalability, thus reinforcing the assertion that Quality of Service (QoS) stands as a pivotal point within the overall spectrum of software development.

Moreover, the report adeptly acknowledges the challenges inherent in scenarios characterized by intricate service dependencies. In this regard, it proffers insights regarding loosely-coupled architectural paradigms as a viable path for moving fast while mitigating QoS cascading and general high-dependent systems issues.

Apart from the core features of applications, the surroundings where systems operate also hold significance. In the article [13], a comprehensive study delves into the effects of different setups involving applications, hardware, and isolation on tail latency for *memcached* (one in-memory key-value distributed cache system) and *Nginx*. The study's main finding highlights that the latency for the same application on diverse platforms can undergo substantial changes, varying by orders of magnitude, depending on factors like application complexity

and system configurations.

The *memcached* application was also used in [48], where the authors analyze the challenges of maintaining high QoS for low-latency workloads in a scenario with shared servers. They state that workload co-location leads to QoS violations due to queuing delay, scheduling delay, and thread load imbalance, and demonstrated it through tests using the *memcached* system. The *memcached* was chosen because it has exceptionally low nominal latency, is sensitive to interference, and has a widely adoption in systems worldwide. By understanding that co-location is necessary because of hardware limitations, the authors proposed some approaches, including modifications in the *memcached* threads scheduler affinity, the usage of general latency-sensitive aware Linux schedulers, specification of CPU bandwidth limits for schedulers, and others. Their conclusions were validated through the collection of metrics, such as queuing delay, scheduling delay, QPS (queries per second), and tail latency, which were achieved through instrumentation in *memcached* and in the operating system.

The conclusions inferred from *memcached* tests of [48] can be extended to the microservices scenario in general. Thus, it is necessary to correctly understand how to distribute services between the hosts, and also monitor them through custom metrics to understand whether QoS violations are caused by the conflict of systems in a multi-system workload co-location scenario. These conclusions are also reinforced by [35], which implements a prototype scheduler for optimizing microservice placement in order to minimize the overall inter-machine traffic.

In instances involving public cloud environments or abstractions like Platform as a Service (PaaS) and Software as a Service (SaaS), direct access to hardware-level details and isolation settings is often restricted. In public clouds, it is common to not know how many replicas a service has, where it is deployed, and whether it is deployed in a single-tenant or multi-tenant model. As it is not possible to scrape the metrics directly from the host, it is important to measure the performance of the target system on the consumer side, inject or plug a supported metrics system. These metrics act as a crucial link between hardware and software abstractions, aiding in detecting and managing potential issues. This role becomes particularly vital due to the limitations inherent in these abstractions.

Continuing with metrics, the article [65] presents one section regarding QoS in microser-

vice architecture. It presents two types of QoS parameters, network-related and infrastructure-related. For network metrics, the ones presented are the number of hops, network healthiness, bandwidth, packet loss, throughput, network delay, and network jitter. For infrastructure metrics, they present the number of CPUs, amount of memory, disk space size, percentage of CPU, and percentage of memory.

Although the metrics presented in [65] are conventional and widely used in generic monitoring, they are not sufficient. For many applications, it is necessary to understand the business context to provide valuable metrics, sometimes entering with metrics in the layer 7 level. For example, in an API scenario, there may be a specific route that triggers a specific flow with an external dependency that could end up causing a QoS violation, and in this case, it is necessary to have information on the average latency per route for the APIs. In the context of a sales application, for another example, it may be necessary to see the average number of orders per minute and notice if it changes suddenly, indicating a possible problem with the flow.

Finally, to help understanding which customized metrics make sense for certain applications, in addition to the general ones, article [76] proposes a Flowchart for identifying and stipulating acceptable values for metrics in the context of cloud computing e-learning applications. Despite the specific domain, the flow can be used as a basis for defining and evaluating metrics for general applications.

2.2.7 Logs and tracing

In the context of a microservices architecture, multiple applications collaborate to compose a system. As previously mentioned, this collaborative behavior introduces various challenges pertaining to inter-application communication.

Within a microservices ecosystem, it becomes imperative to comprehend and visually represent the interactions among these applications. Frequently, scenarios involving failures, latency issues, and unexpected behaviors arise, bringing the need to trace the sequence of requests and investigate each step of their execution.

In this context, two pivotal concepts appears: centralized logging and distributed tracing. Centralized logging confronts the challenge of comprehending and maintaining a historical

record of actions executed during service calls. This serves the purposes of debugging, troubleshooting, and auditing. When a service call traverses multiple systems, each generating individual logs, it becomes crucial to correlate these logs to construct a coherent narrative. Distributed tracing, on the other hand, aims to monitor and visualize the path of requests as they traverse different applications, with a focus on performance optimization and monitoring.

Presently, there exists a multitude of well-established logging solutions. Typically, applications generate logs autonomously and deposit them into files, after which an intermediary entity collects and stores them in a centralized repository. For instance, the ELK Stack[70], comprising Elastic Search, Kibana, Beats, and Logstash, offers a comprehensive suite that facilitates log collection, storage, and visualization. Grafana Loki[46] represents another feature-rich complete logging stack. The logs stored in these solutions inherently contain request identifiers, which can be employed for cross-system log correlation. While logs can also be leveraged to generate metrics and construct monitoring and health dashboards, these functions are not their primary purpose.

In contrast, solving the distributed tracing challenge requires a somewhat more intricate approach. Contemporary solutions typically entail instrumenting application code or introducing agents (or sidecars) to intercept or observe requests in order to assemble request flows. Service meshes, such as ISTIO[72], are frequently employed to address this requirement, but they are not easy to implement and maintain. Another notable solution is Jaeger[73], a widely adopted distributed tracing platform, and was used in the DeathStarBench article.

Still in the realm of logging, tracing, and even metrics, it is essential to highlight OpenTelemetry[75]. OpenTelemetry is an vendor-and-tool-agnostic observability framework and toolkit designed to create and manage telemetry data, standing as a collection of APIs, SDKs, and tools, and being incubated by the Cloud Native Computing Foundation (CNCF). Its primary purpose is to facilitate and standartize the instrumentation, generation, collection, and exportation of telemetry data encompassing metrics, logs, and traces. OpenTelemetry boasts broad availability, spanning multiple programming languages, making it a versatile and well-suited choice for telemetry data management.

Nonetheless, although in early stages, solutions that automate the mapping of distributed

traces are gradually emerging[31][74], with eBPF (extended Berkeley Packet Filter) showing considerable promise in addressing this challenge.

2.3 Resource management

With flexible and volatile environments in the as-a-service model of cloud computing, it is necessary to understand what resource management means. In an ideal world, the resources allocated to a system are used perfectly, without shortages and without waste. However, it is common sense that this is unachievable. It is often necessary to reserve a quantity of unused resources for fluctuations in the use of systems.

The challenge then, considering that resources are expensive, is to try to maximize a system's resource utilization, while minimizing waste, and in such a way that the system is still able to accommodate traffic changes.

Furthermore, within resource management, it is necessary to understand that there are two spectra of management. The first is management within allocated resources, and the other is management of allocated resources. In a cloud computing environment, for example, it is possible to have 3 nodes with 32 vCPUs each and the usage of 60 vCPUS out of the 96 total. It is then necessary to manage which processes the remaining vCPUs will be allocated to, and manage the number of nodes so that there are always free vCPUs in time to be allocated to processes that need them. The example is analogous to managing the number of nodes in a Kubernetes cluster and distributing application pods among the nodes.

The impact of properly allocating resources is sometimes underestimated. However, the literature is growing and some authors are already applying refined algorithms and heuristics in order to better place systems and allocate resources. Some examples of current research include: the aforementioned [35], which proposes a new resource-aware approach to optimize the placement of applications in the cloud environment and presents good research paths; [12], which presents RAA-PI-NSGAII - a multi-objective optimization model that improves the quality and distribution uniformity for the resource allocation problem while balancing different kinds of resource utilization; and finally [51], that brings an overall literature review for resource allocation techniques in cloud computing environments.

2.3.1 Autoscalers

Scaling is the concept of increasing the resources of a system in order to achieve more throughput or maintain system healthiness. Autoscaling, consequently, is the concept of automatically adjusting the resources of a system based on some metrics and target values for it.

There are typically two types of scaling: vertical and horizontal scaling. Vertical scaling involves adding (or removing) resources in a single instance of an application, while horizontal scaling involves adding or removing more instances of one application and distributing the computation between these replicas.

The common scenarios of autoscaling involve periods of high traffic or low traffic, where the system needs to be scaled in order to be able to answer all the incoming requests or be downscaled in order to save costs. A good example is when advertising e-commerce sites on television, where access peaks are reached and require the system to be able to scale up and adapt to the new scenario quickly, or else sales are directly affected.

In the context of monoliths, the scaling is simple, it is possible to achieve good results by increasing the resources of the monolith instance or adding new instances and distributing the traffic. On the other hand, as the entire system is one single application, its scaling implies scaling all the features and parts, even the ones that are not being used.

In the context of microservices, some implicit challenges appear. As applications depend on each other, sometimes the hotspot is a false positive or is being caused by a direct or indirect dependency. Scaling the wrong part of the system can exacerbate the unsaturation and cause massive cascade effects, as exemplified in earlier sections.

Still, regardless of the architecture used, the metrics commonly used nowadays mainly look at resource usage. CPU and RAM usage are widely used for scaling arbitrary systems and are the default metrics for big frameworks and products, such as Kubernetes[40] and Amazon Web Services (AWS) EC2[3].

3 Research Development

Understanding the complexities introduced by the microservices architecture is crucial to maintaining reliability and availability in modern distributed systems. As these systems grow in scale and modularity, the interdependence between services becomes more dynamic and more difficult to manage. Visualizing service communication is therefore essential for debugging, performance optimization, root-cause analysis, and security auditing.

Despite the growing ecosystem of observability tools, most solutions still require manual instrumentation, sidecars, or significant changes to the application code or infrastructure. This can be labor-intensive, error-prone, and impractical at scale. Furthermore, many approaches rely on sampling or partial tracing, leading to incomplete or inaccurate representations of service interactions.

In this context, this research proposes a novel, low-overhead, and non-intrusive approach to mapping service dependencies in Kubernetes-based microservice environments. Using Cilium — a network and security observability platform built on eBPF — and its Hubble extension, this work demonstrates how to extract real-time Layer 4 and Layer 7 communication data to build dynamic dependency graphs between services.

The proposed methodology eliminates the need for traditional code instrumentation or proxy sidecars and instead captures raw network flows directly from the kernel level. Using these data, the system computes degrees of direct and indirect dependencies and models how services interact under different operational conditions.

By validating this approach using open-source microservice benchmarks and controlled experiments, the study aims to show that dependency graphs generated from network observability data are accurate and actionable. This contributes to the broader field of microservice observability by providing an automated, scalable, and adaptable method for real-time dependency analysis — laying the groundwork for future research in system visualization, anomaly detection, impact analysis, and topology-aware optimizations.

3.1 Hypothesis and Objective

The hypothesis and the objective of the work can be described as follows:

Hypothesis: *It is possible to automatically and dynamically map service dependencies in microservices-based architectures deployed on Kubernetes by leveraging clusters network infrastructure tools such as Cilium and Hubble, without requiring intrusive instrumentation, and obtain meaningful insights into system behavior that may facilitate service management and monitoring.*

Objective: *To develop and validate a non-intrusive methodology for mapping dependencies between microservices in Kubernetes clusters using Cilium, analyzing the collected data to compute dependency degrees, and demonstrating the feasibility of this approach for improving observability and understanding of complex microservices environments.*

3.2 Research context

As the name implies, microservices are famous for handling small logical portions of large systems; therefore, large systems can be made up of several microservices that communicate with each other. One of the advantages of using microservices is the ability to scale specific parts of the system [24], instead of the entire system, and save resources. However, this advantage can be considered as a double-edged sword because, with a microservice architecture, the complexity of knowing which part to scale and when to scale increases considerably.

Section 6 of the article "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems" [28] gives a good introductory example for the theme:

> *Microservices complicate cluster management, because dependencies between tiers can introduce backpessure (sic) effects, leading to system-wide hotspots (...). Backpressure can additionally trick the cluster manager into penalizing or upsizing a saturated microservice, even though its saturation is the result of backpressure from another, potentially not-saturated service. Fig. 17 highlights this issue for a simplified two-tier application consisting of a webserver (nginx), and an in-memory caching key-value store (memcached). In case A, as the client issues read requests, nginx reaches saturation, causing its latency to increase rapidly,*

and long queues to form in its input. This is a straightforward case, which autoscaling systems can easily tackle by scaling out nginx, as seen in the figure at t = 14s and t = 35s.

Case B on the other hand, highlights the challenges of backpressure. When using HTTP1, requests within a single connection are blocking, i.e., there can only be one outstanding request per connection across tiers. Therefore, even though memcached itself is not saturated, it causes long queues of outstanding requests to form ahead of nginx, which in turn cause it to saturate. Current cluster managers cannot easily address this case, as a utilization-based autoscaling scheme would scale out nginx, which is busy waiting and appears saturated. As seen in the figure, not only does this not solve the problem, but can potentially make it worse, by admitting even more traffic into the system. Even without the connection blocking in HTTP1, backpressure still occurs, as multi-tier applications are not perfect pipelines where tiers operate entirely independently.

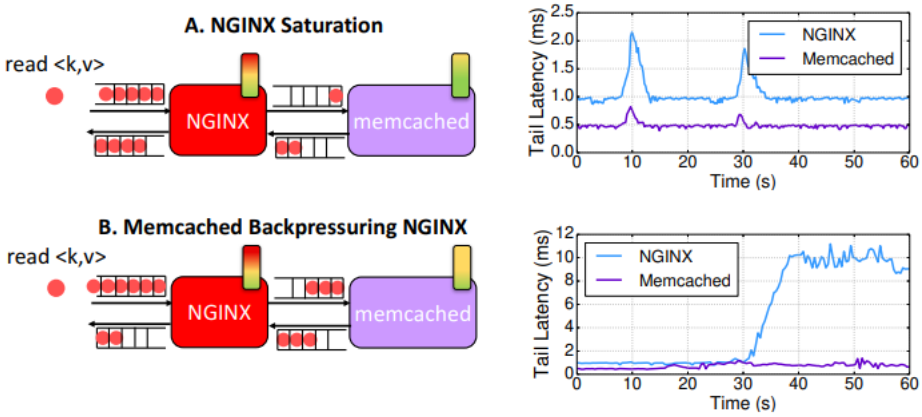


Figure 9: *"Example of backpressure between microservices in a simple, two-tier application. Case A shows a typical hotspot that autoscalers can easily address, while Case B shows that a seemingly negligible bottleneck in memcached can cause the front-end NGINX service to saturate".* (Figure 17 of [28]).

In the example above, small oscillations in a system composed of two small applications can cause expressive problems not only for current autoscalers but also for the entire system's

healthiness. When trying to manage the problem, the autoscaler may end up causing more complications for the system since it may scale a wrong component and cause even more traffic to be admitted to the system or to the saturated component.

In a hypothetical scenario, consider a system composed of two applications, denoted as A and B. Each application is equipped with its respective autoscaler, which employs a CPU target rule set at 70%. Additionally, there exists a partial dependency from application A to application B. However, understanding the implications on application A when application B is under performance pressure is not a straightforward task, and understanding the communication pattern between them is also difficult, since they can change depending on various external conditions [50]. As demonstrated in the above example, it becomes apparent that scaling application A before addressing the performance issues of application B can potentially exacerbate the overall system's condition. Therefore, it is crucial to gain a deeper understanding of the possible implications arising from the interrelationship between the microservices. This entails considering the intricate relationship between applications A and B, rather than solely relying on CPU metrics, prior to initiating any scaling actions.

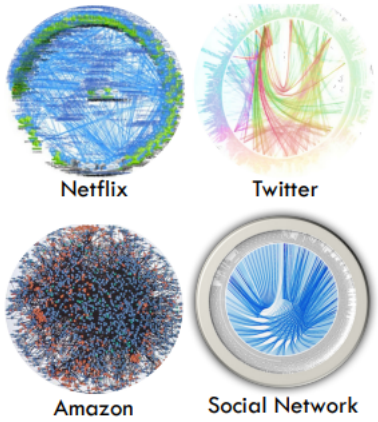


Figure 10: *"Microservices graphs for three real production cloud providers [...]"* (Figure 18 of [28]).

The two aforementioned examples provide valuable insight into the implicit challenges related to scaling and managing microservices. However, it is crucial to acknowledge that these examples remain considerably limited in comparison to the genuine issues encountered in practice. As depicted in Figure 10, the microservice graphs of large systems, such as Net-

flix, Twitter, Amazon, and one of the systems designed by the author, which emulates a small social network, present a more complex representation. As the number of applications (microservices) within an application grows, the overall complexity of the system escalates to a level wherein the communication graph between services becomes excessively vast, making it challenging to analyze. In such circumstances, the availability of tools capable of comprehensively and centrally interpreting the system becomes imperative. These tools are essential for making informed decisions about system and resource management and application scaling.

Mapping communications within a system remains a non-trivial task, and static mappings do not capture fluctuations in the Microservice Call Rate (MCR), as mentioned in [50]. Although various technologies are available to facilitate mapping activities, including OpenTelemetry [75] and Service Meshes such as Istio [72], they do not natively elaborate on the implications of dependencies between services and require implementations and instrumentations in each application of the system – making the task labor-intensive.

On the other hand, as exemplified by CNIs, most virtualization systems use virtualized networks, which play a crucial role in managing packets and facilitating communications between different components of the system, and can be used for mapping systems in a microservices scenario.

Given the aforementioned considerations, it is evident that substantial progress is yet to be made in mapping and understanding the multifaceted implications associated with the utilization of microservices. Therefore, this work seeks to: (1) use the microservice environment to provide an approach to mapping the communications between the applications; (2) prove that the proposed mapping strategy brings relevant insights for microservices management, opening space for further research in scaling, security, and other areas.

3.2.1 Related work

The preceding sections have demonstrated the novelty surrounding the topic of microservices, and, as previously contextualized, their challenges continue to be under investigation. Despite the abundance of articles discussing the characteristics of microservices, particularly concerning the inherent communications and networking issues, only a limited number of articles have addressed the correlation between these challenges and the management of mi-

croservices. In addition, these works neither infer and present it in a visualizable way nor explicitly directly use the relationship between services.

The article entitled "Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices"[80] addresses the same scaling problems outlined earlier, which pertain to the dependencies between microservices causing backpressure effects and cascading QoS violations within the entire environment. Sinan serves as a data-driven cluster manager for interactive microservices, operating in an online and QoS-aware manner. It employs a set of machine learning models to analyze and predict the performance impact of inter-microservice dependencies, thereby enabling appropriate resource allocation and preservation of end-to-end tail latency.

Sinan also emphasizes the inherent limitation of current cluster managers designed for monolithic systems, as they do not grasp the intricacies of microservices. To overcome this limitation, the approach adopted by Sinan involves the utilization of two artificial intelligence models: one for short-term performance prediction and the other for long-term performance evolution. Additionally, a centralized resource manager, coupled with per-node resource agents, is employed to monitor application performance and resource utilization across the cluster. Using ML models, Sinan accurately predicts the end-to-end latency and the likelihood of a QoS violation for a given resource configuration, based on the system's current state and historical data. These predictions are then used to optimize resource efficiency while meeting QoS requirements. The model demonstrated its effectiveness in predicting system oscillations and maintaining end-to-end latency within QoS expectations, without any violations, during the tested scenarios.

Sinan's authors, some of whom had previously co-authored DeathStarBench, employed two DeathStarBench systems (Hotel Reservation and Social Network) during the development and testing of their model. These systems were divided into three distinct layers: frontend, business logic, and caching & db layer. The primary focus of their investigation was on the allocation of CPU resources to each tier, accomplished through Linux cgroups via the Docker API. Each tier was provisioned with the maximum profiled memory usage to effectively mitigate memory-related errors.

The authors also present several management challenges that justify the use of a Ma-

chine Learning approach. Firstly, they note that the presence of dependencies among tiers further complicates resource management. Microservices, being more than simple pipelines, can introduce backpressure effects that are difficult to detect and prevent effectively. Secondly, they argue that the complexity of the system makes empirical approaches significantly more difficult and unfeasible, particularly in complex microservice scenarios involving numerous dependent applications. Third, they highlight the delayed queueing effect inherent in microservices scenarios, which necessitates proactive prevention by the resource manager to avoid potential QoS breaches – acting promptly is crucial to prevent adverse scenarios. Lastly, they emphasize the data complexity within a microservices environment, wherein the adoption of Machine Learning offers the advantage of reducing exploration overhead. Using ML, the resource data space can be effectively scanned to meet requirements, mitigating the uncertainties and inefficiencies associated with empirical approaches. In conclusion, their decision to employ ML is motivated by the limitations of empirical methods, which can lead to unpredictable performance and resource inefficiencies.

Artificial intelligence models are often perceived as black boxes that solve problems in ways that lack full comprehension. To address this concern, the authors introduced a method for interpreting the output of the ML model by employing LIME [59]. LIME was utilized to fit a linear regression model, enabling a better understanding and interpretation of the ML model's behavior. However, it should be noted that this approach to comprehending the scenario proved to be relatively expensive and not straightforward to implement.

Although Sinan has showcased commendable scaling outcomes and microservices dependency insights, the proposal presented in the current work embodies several distinct approaches.

Initially, Sinan does not perform the automatic mapping of dependencies and communications. Instead, the deep correlation between system applications is inferred implicitly and automatically through machine learning models. The authors supply the artificial intelligence model with the system's topology and the manually collected performance data of the applications. However, this approach poses certain issues, particularly in instances where the system's topology changes, new applications are introduced, or alterations in applications lead to adjustments in the manually provided model parameters, among other

potential scenarios.

Secondly, it is essential to acknowledge that the utilization of linear regression techniques to gain insights into the model's behavior post-training is a non-trivial and not full precise task. While the authors attempt to address the "black box AI" challenge through linear regression techniques in the model, these methods remain characterized by some significant limitations: (1) they are not easily comprehensible; (2) they entail non-trivial challenges in their application and replication; and (3) they do not entirely capture the inherent unpredictability exhibited by artificial intelligence models.

Lastly, it is worth noting that in Sinan, services are deployed utilizing Docker Swarm, with significant metrics collected directly through the Docker API. Presently, Kubernetes has gained widespread popularity, surpassing Docker Swarm, particularly in public cloud environments. Consequently, gathering metrics directly from the host machines of the deployment platform becomes unfeasible - often, pertinent details regarding the operating system utilized by public clouds for running containers remain undisclosed. Furthermore, the authors' choice to allocate spare memory to the tiers as a precaution against memory errors and their reliance solely on CPU metrics for decision-making have raised certain concerns, as numerous studies have deemed these metrics as outdated within the microservice context, advocating for the adoption of more efficient and relevant alternatives.

After considering all the aforementioned aspects, it is evident that there exist substantial differences between Sinan and the approach presented here. Despite this, it is important to acknowledge the merit of Sinan's work, as it introduces several innovative aspects. Therefore, it would be of considerable interest for future endeavors to explore the possibility of leveraging Sinan's artificial intelligence model along with the model and data presented in this study to observe its behavior and outcomes. Such an approach holds promise for advancing the field and gaining valuable insights.

"Overload Control for Scaling WeChat Microservices"[81] is another article that addresses the issue of resource management and microservices dependencies. The study introduces DAGOR, a microservices-specific overload control scheme specifically designed to be service-agnostic, independent, efficient, and fair, built for WeChat microservices.

The article defines a scenario where requests along the call path fail as an overload

scenario, while subsequent overload refers to coexisting multiple overloaded services or a single overloaded service repeatedly invoked by its associated upstream services in the microservices context, leading to degraded system throughput and increased challenges. Detecting overload is non-trivial, and to address this, the authors of DAGOR utilize the average waiting time of requests in the pending queue to profile application load status, and caution against relying on CPU utilization alone for detecting overload, as high load does not necessarily indicate overload. They advocate considering other indicators in addition to CPU utilization to accurately identify overload scenarios.

Upon detecting an overload, the authors highlight the issue of allowing microservices to handle the overload independently, given the complexities arising from service dependencies. To address this challenge, they implemented a service admission control mechanism incorporating two types of priority-based admission control, along with adaptive and collaborative admission control techniques, to ensure the overall healthiness of the system. As a result of their successful implementation, DAGOR has been effectively utilized for over five years in the WeChat business system at the time of the article.

While the overload control technique has demonstrated considerable interest and efficiency in the presented scenario, it must be acknowledged that its implementation also possesses certain limitations. Given its nature as a control mechanism, it inevitably entails prioritization and regulation of traffic, potentially leading to adverse consequences for specific requests. A notable instance in the referenced article illustrates the situation where login actions are granted higher priority than sending a message with an image, thereby potentially causing delays or unresponsiveness for the latter operation.

Despite the automatic assembly of graphs depicting communications between services, the article lacks explicit mapping and utilization of dependency relationships among the services. Instead, the systems appear to be categorized based on their call path to facilitate the sharing of business priorities for requests. Microservices, in turn, rely on these assigned priorities to guide their functioning. Consequently, communication between systems is indirectly used as priorities are passed from one service to another.

The presented model demonstrates a strong emphasis on addressing WeChat's specific business requirements. Although it claims to be "wechat service's agnostic" and extendable

to other services, achieving such generality is not a straightforward endeavor. It is worth noting that implementing the model for other systems may necessitate tailored actions and considerations unique to each type of system. The article on DAGOR also employs simplistic metrics, which work for the WeChat scenario but can generate several false positives in other scenarios. Therefore, enhancing the model with more sophisticated and service-specific metrics can significantly contribute to making well-informed decisions.

These aforementioned articles are the ones that most cite points in common with the ones to be presented in this work. Even so, the literature contains numerous other solutions for handling fine-grained large-scale resource management, QoS-aware resource management, admission control tools, ML-based resource management, and QoS solutions [57] [29] [55] [61] [58] [79] [21] [20] [23] [49] [53] [63] [66] [22] [16] - among others previously cited. However, these solutions do not specifically address the inherent complexity in microservices communications, which remains a significant challenge in the field. The unique intricacies of microservices architectures demand specialized approaches that account for microservices characteristics using them in a centralized and globally aware decision-making strategy.

3.3 Dependencies mapping

As previously mentioned, the microservice architecture, which promotes a high granularity of services within a system, introduces challenges related to interdependencies among systems.

Automatically mapping the inter-service relationships and determining their call rates in service-based systems is a complex task, as the behavior of the system may vary dynamically. Although the same services often appear in different critical flows, static mappings are not sufficient for granular information due to Microservice Call Rate (MCR) fluctuations, as mentioned in [50]. The behavior of these systems can vary significantly depending on various external conditions, such as the time of day or even a growing demand caused by a live television advertisement.

Understanding what are the connections between services and how they interact is of absolute importance to prevent and anticipate possible failures, solve problems, and maintain the observability of system workflows. Dependency mapping is a subset of the distributed tracing concept, which emerged together with a trend for systems to become more complex

and distributed.

The importance of system observability is underscored by the numerous initiatives aimed at addressing this challenge. Several companies have been created to address these challenges, such as Datadog, New Relic, and Dynatrace. However, the community also developed several open source tools to improve observability and manage system complexity, such as Jaeger, Grafana Tempo, SigNoz, Zipkin, Istio, LinkerD, and the OpenTelemetry ecosystem.

3.3.1 Traditional tools for dependencies mapping

Before the advent of eBPF technology, dependency mapping and management within the microservices context were pursued through different methods and tools. Some of the most used approaches included service meshes, manual instrumentation, request-ID logging, and distributed tracing with tools like Linkerd and Istio.

A service mesh is an infrastructure layer that focuses on making services communicate with each other. It frequently uses a sidecar proxy pattern. Among others, notable service mesh solutions provided in-depth features related to load balancing, service discovery, and failure recovery, which are critical functionalities of microservices dependency governance.

Being one of the first service meshes, Linkerd already allowed users to visualize service dependencies and monitor the performance of applications. Thanks to built-in telemetry, the proxy injected beside each microservice lets Linkerd capture traffic data and give insight into relationships and health without user changes in the application code.

Istio, one modern Service Mesh, was built on top of the ideas introduced by Linkerd. Istio added powerful features for policy enforcement, traffic management, and security. The observability capabilities of Istio allowed users to collect metrics, logs, and traces in a way that can map dependencies between services. With Istio's use of the Envoy proxies, it was possible to monitor traffic effectively and create a dependency map across the microservice architecture.

In the past, with the dawn of automated solutions, microservice developers had to deal with manual instrumentation, where they were expected to instrument all services with special lines of code to make logs of specific events, monitor function calls, and record interactions with other services. Although very precise and detailed, this process is extremely laborious

and subject to many human errors. In turn, maintaining the instrumentation code, in numerous services, came with many challenges and turned into a truly time-consuming and tough process.

Another pattern that allowed developers to map the flow of the requests across the distributed system is the "request-id" logging. With a unique identifier to identify each request, it became possible to trace its path across the different microservices. The common practice is to generate a unique request ID supposedly at the entry point of the system and include it in the logs of each service when processing that request.

Distributed tracing is an important idea for mapping how services depend on each other. It involves adding code (instrumentation) to each service so that they send trace data about their performance. Tools like Jaeger or Zipkin collect these data and show how requests flow, how long they take, and which services interact with each other. Each service sends its trace data to a central server, which can then create diagrams to showcase details of end-to-end dependencies and performance.

In commercial products, tracing support was provided as part of large observability platforms, including products like Datadog, New Relic, and Dynatrace. These tools provide access to advanced features, possible automated instrumentation, advanced analytics, and capabilities to work with other monitoring systems.

3.3.2 Using eBPF in microservices management

eBPF is a revolutionary technology that allows users to run programs in the kernel space of Linux without modifying the kernel source code or having to add new kernel modules. Born as an approach to packet filtering, it evolved into a versatile toolbox that enables the safe, sandboxed execution of programs at the heart of the OS kernel. eBPF is a secure, sandboxed kernel-space environment for running small, verified programs, which has opened up a new world for fine-grained monitoring, tracing, and control over networks.

eBPF programs are usually written in some subset of C and compiled into bytecode, which the eBPF virtual machine can execute in the kernel. At this level, it is possible to attach these programs to different kernel hooks, such as system calls and network events, and tracepoints can be used to allow observability and control for many different types of

operations.

In the context of microservices applications, eBPF provides many new opportunities and can be used for activities such as:

- Enhance kernel-level visibility: eBPF enables an unparalleled level of visibility into kernel-level events, enabling full understanding of interactions among microservices. It becomes possible to performatically trace the flow of the entire request and gather as much information on every interaction by attaching eBPF programs to every single network socket and every system call made during runtime.

- Dynamic/automatic instrumentation: unlike the classic methods that stipulate manual instrumentation of code, eBPF allows dynamic instrumentation. It means that developers can insert eBPF programs into running systems without service restart or modifications to application code. This greatly reduces the overhead and complexity of manual instrumentation.

- Monitors in low-level monitoring: eBPF is kernel-based and its data collection has very low performance overhead. This makes eBPF a vital tool in very high-throughput, sensitive-latency microservices environments where traditional monitoring tools could bring big overheads.

- Collect real-time data: eBPF allows the capture of real-time analytics data, hence providing real-time visibility into service interdependencies, performance, and so much more. These real-time features are required for these dynamic, ephemeral environments where most service architectures reside.

- Security and segregation: sandboxing eBPF programs ensure they do not destabilize or compromise the security of the system; therefore, the use of eBPF for monitoring and tracing in a production environment can be secured.

3.3.3 Cilium and Hubble

Cilium is an open-source software that allows monitoring, securing, and enabling the networking of container workloads. The software was designed specifically for Kubernetes

environments, but can be modified for other container orchestration platforms. eBPF is the core technology under the hood of Cilium, which is rapidly evolving, with new features emerging as its community and adoption expand.

eBPF allows Cilium to programmatically inject networking, security, and performance monitoring logic on Linux systems without modification to kernel code or additional modules as a convenient, minimal-overhead way to deeply integrate the logic at various points of the network stack. With eBPF, Cilium enforces network policies, balances load requests, enables custom monitorings, and logs network behavior.

In addition to providing high performance, custom security policies, and flexibility for extensions, Cilium and its ecosystem offer tools that can facilitate the task of mapping the communication between microservices. For this reason, it was used as the CNI of the Kubernetes environment in the development of this work.

The core features of Cilium are the following:

- Network policy enforcement: Cilium provides fine-grained access control defining what endpoints are allowed to communicate with each other in a network. These are implemented using eBPF, which ensures high-performance packet filtering and policy execution.

- Load Balancing: eBPF on Cilium allows for the insertion of load balancing directly into the Linux kernel, thus enhancing performance and reducing latencies when compared to conventional user space or proxy-based solutions.

- Network Security: eBPF enables fine-grained visibility into network traffic at a packet level, making this capability highly effective for assisting in the real-time identification and blocking of anomalies and threats.

Cilium also brings the concept of Cilium Flow, which refers to basic network communication events monitored and manipulated by Cilium, and captures network data at Layer 3 and 4 (ISO/OSI), such as source and destination IP, addresses, protocol, and ports. Using the Cilium Flow concept, the software is able to provide fundamental visibility and enforce network policies within the Kubernetes network environment.

The cilium ecosystem also includes other tools, such as Hubble[14], which is a fully distributed networking and security observability platform for cloud-native workloads built on top of Cilium and eBPF. By its own definition, Hubble "enables deep visibility into the communication and behavior of services as well as the networking infrastructure in a completely transparent manner", and can be used for the desired mapping of inter-service communications. It provides a platform of network visibility and security analytics, sized and designed to the scale and complexities that modern container networking involves. It leverages the rich Cilium Flows information collected by Cilium through eBPF and expands it to offer deep insight into what is happening in the network, introducing the Hubble Flow concept.

Hubble Flows, in addition to the Cilium Flows information, contains also Kubernetes-specific information like pod and namespace details, as well as application layer (L7) data, such as HTTP requests. Thus, Hubble Flows offers an even more richer and comprehensive view of network traffic for advanced observability and security.

Hubble's features include:

- Real-time network flow visualization: Hubble visualizes network flows in real-time, which enables understanding the communication patterns and dependencies of applications running in their cluster.

- Service dependency mapping: Hubble provides insight into how different services at any given time are communicating with each other, providing critical support for both the ability to troubleshoot and optimize application performance.

- Security observability: Hubble provides complex telemetry data that can be very helpful in identifying potential security problems, such as unexpected connection attempts, protocol violations, and more.

Together, eBPF-powered Cilium and Hubble provide a full-stack extensible solution for monitoring and securing microservices architectures, delivering deep visibility into both network and application performance.

3.3.4 Emerging tools for automatic mapping

It is important to note that the state-of-the-art in service dependency mapping is evolving rapidly. During the development of this research, two notable tools have emerged that contribute to the observability and understanding of service interactions in distributed systems: Coroot[6] and Deepflow[7].

Coroot is an open-source observability tool that integrates eBPF-based telemetry collection with a user-friendly interface to analyze the root causes of performance issues. It correlates metrics, traces, and logs to build a real-time map of service dependencies and latency bottlenecks. The core objective of Coroot is to provide a proactive approach to troubleshooting by allowing developers and operators to visualize how services interact, identify performance regressions, and trace anomalies to specific root causes at the service level without requiring manual instrumentation.

Deepflow, on the other hand, focuses on combining traditional observability with network-aware insights. It leverages eBPF to capture both L4 and L7 traffic and enriches the data with workload metadata to build service maps. Deepflow aims to bridge the gap between DevOps and NetOps by providing a unified platform to observe network flows, diagnose dependencies, and correlate network behavior with application performance. Its approach emphasizes automated topology reconstruction and contextual analysis across cloud-native environments.

Despite the similarities in using eBPF and providing dependency graphs, this research proposes a distinct and complementary approach. While Coroot and Deepflow are primarily concerned with diagnostics and visibility, this work introduces a non-intrusive, real-time methodology focused on extracting dependency data to support system-level decision-making, particularly in the context of topology-aware management and service importance assessment. By computing direct and indirect dependency degrees and validating their relationship with system behavior, this research opens the door to dynamic resource management strategies based on the actual communication patterns of microservices. Thus, the contribution goes beyond observability, aiming to integrate network-derived insights directly into the lifecycle of systems management.

3.4 Degrees of dependencies

As mentioned above, Hubble is the observability platform within Cilium. With Hubble, it is possible to observe the flows in the cluster. In this context, a flow is a record of network traffic that captures the metadata of the packets traversing the network, such as source and destination IP and ports, protocols, packet sizes, etc. Hubble is built on top of the Cilium Monitor and uses it to provide real-time insights into the communication patterns within Kubernetes clusters.

Cilium Monitor is a Cilium core component that, using eBPF, captures and displays detailed information about network events in real time. Hubble registers a listener on the Cilium Monitor and stores the listened events in a structure called Hubble Ring Buffer. The ring buffer can store a set number of events in memory. When the ring buffer becomes full, the older events are overwritten by the new ones.

With a simple code in Golang, it is possible - through the Hubble gRPC Service - to read data from Hubble Ring Buffer. The example of the code is presented in the Appendix B. The flows read in the Ring Buffer contain the full flows data, including the source and destination services, the protocol with the flags (e.g. TCP ACK), and Layer 7 data when available. Also, it is possible to specify "Flow Filters", which enables more specific data filters. Therefore, by subscribing to the Hubble Ring Buffer it is possible to listen to the network flows and use the data to map service dependencies.

Besides usging the Cilium Ring Buffer, it is also possible to obtain information about the flows via Hubble Metrics and via the Hubble flows Exporter. Hubble Metrics are a set of metrics collected by default and that can be used for observability purposes. Two metrics that can be interesting for the purposes of this research are the *hubble_tcp_flags_total* and the *http_requests_total*, the latter being restricted to HTTP applications. These metrics count the absolute values of TCP flags and HTTP requests in the cluster network. However, Hubble supports the addition of more labels to the metrics, called Context Options[71], which enable the addition of a set of labels including *source_workload* and *destination_workload*. Thus, it is possible to arrive at the same data as the Ring Buffer subscription approach by configuring the Hubble metrics and scraping them from the Cilium agents.

Finally, Hubble Exporter is another way to perform mapping via Hubble. The exporter is

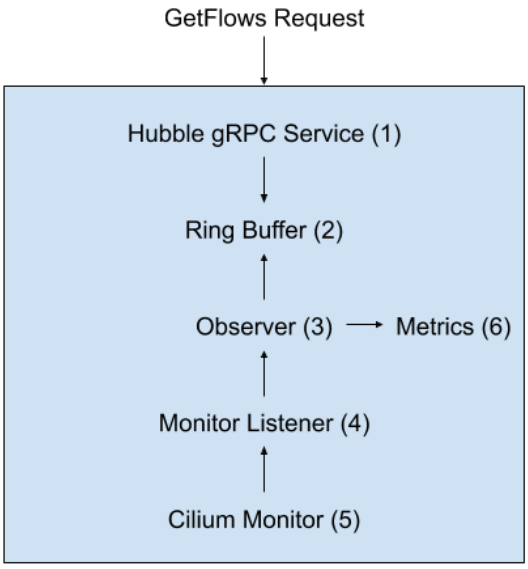


Figure 11: Hubble Architecture. Image from the official website.

a recent feature that allows exporting/writing network flows information into files persisted on disk. Hubble Exporter has file rotation and size limits features, together with filters and field masks. Filters and field masks can be used to select information and improve disk usage and processing time. Once the data are stored on disk, some external processing mechanism can consume the events and build the dependency mapping in real-time - through watching the file - or offline.

Although there are some ways to retrieve the flow data, the mapping itself is also made up of parsing and understanding these data through some algorithm. For OSI Layer 4 (transport layer) network mapping, the nature of applications should guide the selection of the algorithm to use. Additionally, many applications only use HTTP to talk to each other, and each HTTP request may require a unique TCP connection. Then, if the TCP keepalive mechanism is disabled, it is possible to filter all TCP SYN flows between the services and then build the graph of connections in real time. As the number of connections between the services can be retrieved from the number of TCP connections with these configurations, it is possible to estimate it by the number of SYN flags exchanged by the services.

Disabling the emission of TCP keep-alive messages, while potentially facilitating the anal-

ysis of ephemeral connections, can adversely affect the reliability and performance of environments. For instance, the absence of keepalive messages may result in undetected dropped connections or the premature teardown of idle links, compared to configurations where keepalive is enabled. It is important to note, however, that keepalive also has drawbacks, and not all environments use it.

As seen in the mapping approaches, Hubble also supports Layer 7 introspection, which is interesting for dependency mapping, since it enables accurate measurement of total requests between services. For that, Cilium uses an embedded Envoy proxy (that can become dedicated) in each Cilium Node Agent, which is responsible for enforcing network policies and providing connectivity and observability for the cluster workloads. The traffic is forwarded through the Envoy proxy, in a model similar to a service mesh sidecar, but which runs at the cilium agent level and not at the workloads' pod level. Currently, Hubble Layer 7 observability supports and can show data for DNS, HTTP, and Kafka protocols. However, the community is large and is working on expanding the protocols, as the functionality was well-accepted.

In future work, alternative approaches could be considered to enable the use of TCP keepalive messages and enable more Layer 7 protocols, including direct flow counting, leveraging other Layer 4 indicators, or incorporating solutions such as done by Coroot[6], which trace TCP connections by using the `sys_connect` and `inet_sock_set_state` Kernel tracepoints together with eBPF to discover TCP connections and LISTEN sockets, and trace application-layer protocol requests by monitoring `sys_write/writev/sendto` and `sys_read-/readv/recvfrom` tracepoints. These strategies require further investigation and fall outside the scope of the current work.

3.5 Dependency analysis

Having access to the requests exchanged between applications enables the identification of service dependencies and the quantification of their interdependence. This, in turn, facilitates the assessment of how one service can influence or impact another.

To quantify the degree of dependency between services, a mapping model is proposed and implemented, utilizing the previously collected data.

3.5.1 Modeling

For the proposed modeling, consider the following definitions: a system is composed of N ($N \geq 1$) applications, where an application is denoted as X_i ($0 < i < N$). In a microservices scenario, there is a high level of communication and dependency between applications, and the degree of dependence of an application X_i on an application X_u is denoted as $S(X_i, X_u, d, t)$, where d is the maximum depth of dependencies to be considered and t is the number of seconds considered in the calculation.

In the given context, the following notation is defined:

- $\alpha(X_i, t)$ represents the number of requests (calls) that application X_i received in the last t seconds.
- $\beta(X_i, X_u, t)$ represents the number of requests that application X_i sent to application X_u in the last t seconds.
- $\mu(X_i, X_u)$ represents the minimum number of hops (applications) required for X_i to reach X_u in the (to be defined) dependency degree graph.

The time window t serves as a sliding aggregation parameter that dictates the temporal granularity of the metrics used. By shortening t (for example, to 30s), the model can become more reactive, emphasizing sudden bursts or anomalies – an advantage for real-time alerting or autoscaler triggers. In contrast, increasing t (for example, 30min or 1h) decreases transient noise and exposes persistent communication patterns, which may be better suited for capacity planning or architectural refactoring. Because the same t applies consistently to both α and β, adjusting its value offers a straightforward way to trade immediacy for stability without altering the fundamental structure of the dependency calculations.

The second key parameter is the maximum depth d, the number of hops the algorithm is allowed to traverse when following call chains. Choosing $d = 1$ limits the analysis to direct caller–callee pairs, which is ideal for simpler graphs with straightforward conclusions. Raising d progressively incorporates indirect dependencies, capturing how latency, errors, or saturation can propagate across multiple tiers. This broader view is valuable for resilience

engineering, chaos testing, and "blast-radius" estimation, but it also enlarges the graph and can amplify noise from low-traffic edges and complicate the analysis.

The formula for determining the degree of dependence between application X_i and application X_u, denoted as $S(X_i, X_u, d, t)$, is defined as follows:

$$S(X_i, X_u, d, t) = \begin{cases} \frac{\beta(X_i, X_u, t)}{\alpha(X_i, t)}, & \text{if } X_i \text{ directly communicates with } X_u, \\ 0, & \text{if } d < \mu(X_i, X_u), \\ \sum\limits_{\text{paths}} \left(\prod_{X_w} S(X_i, X_w, d-1, t) \cdot S(X_w, X_u, d-1, t) \right), & \text{if } X_i \text{ does not directly communicate with } X_u, \\ & \text{and } d \geq \mu(X_i, X_u) \end{cases}$$

A system can then be represented as a bidirectional weighted graph – the dependency degree graph –, where each vertex is an application and each edge leaving a vertex X_i and arriving at a vertex X_u represents the degree of dependence between X_i and X_u.

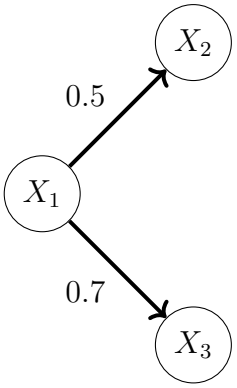
Thus, in a system composed of three applications with the following data points (metrics):

$$\begin{aligned} \alpha(X_1, t) &= 10 \\ \alpha(X_2, t) &= 20 \\ \alpha(X_3, t) &= 30 \\ \beta(X_1, X_2, t) &= 5 \\ \beta(X_1, X_3, t) &= 7 \end{aligned}$$

Considering $d = 1$, it is possible to infer that:

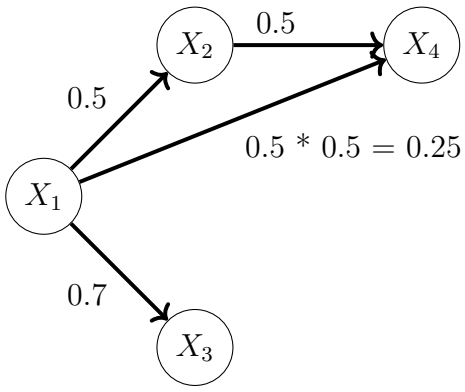
$$\mu(X_1, X_2) = 1$$
$$\mu(X_1, X_3) = 1$$
$$S(X_1, X_2, 1, t) = \frac{\beta(X_1, X_2, t)}{\alpha(X_1, t)} = \frac{5}{10} = 0.5$$
$$S(X_1, X_3, 1, t) = \frac{\beta(X_1, X_3, t)}{\alpha(X_1, t)} = \frac{7}{10} = 0.7$$
$$S(X_2, X_3, 1, t) = \frac{\beta(X_2, X_3, t)}{\alpha(X_2, t)} = \frac{0}{20} = 0$$
$$S(X_2, X_1, 1, t) = \frac{\beta(X_2, X_1, t)}{\alpha(X_2, t)} = \frac{0}{20} = 0$$
$$S(X_3, X_2, 1, t) = \frac{\beta(X_3, X_2, t)}{\alpha(X_3, t)} = \frac{0}{30} = 0$$
$$S(X_3, X_1, 1, t) = \frac{\beta(X_3, X_1, t)}{\alpha(X_3, t)} = \frac{0}{30} = 0$$

And represent it as



where omitted edges represent the value 0.

To clarify the mapping of indirect dependencies, the example below adds a fourth X_4 application. Therefore, if X_2 communicates with X_4 with $S(X_2, X_4, t) = 0.5$ and we increase d to 2, we would have:


X_2
0.5
X_4
0.5
0.5 * 0.5 = 0.25
X_1
0.7
X_3

4 Experiments and Results

This section presents the experimental setup, the methodology adopted for data collection and analysis, and the results obtained during the evaluation of the proposed approach, aiming to validate its effectiveness in real-world microservices environments.

4.1 Test environment: infrastructure

For the development of this work, a microservice-based application execution environment is necessary to carry out the analysis and tests. Nowadays, several platforms are used to deploy microservices, ranging from bare-metal deployments to container-based and serverless deployments.

Due to the nature of microservices, container-based deployment architectures have become widely used to run microservice systems. Kubernetes is an open-source orchestration system for containers and is currently considered state-of-the-art for deploying container-based systems. Spotify [43], Yahoo [45], IBM [39], Nokia [42], Booking [38], OpenAI [41], BlaBlaCar [44] and many other well-known companies maintain planetary systems on top of Kubernetes, which proved to be efficient enough to run the most diverse use cases.

We used Kubernetes as the platform for developing the work. The choice for Kubernetes, in addition to it being highly known and globally adopted, was made due to the fact that it provides a complete platform to carry out deployments. Almost always, systems can be easily ported in and out of Kubernetes, making conclusions drawn from Kubernetes easy to generalize to other deployment platforms.

One Kubernetes cluster was provisioned in Magalu Cloud [32] using the Magalu Cloud's managed Kubernetes as a Service. The cluster contained one Node Pool, with 5 virtual machines, each one with 8 vCPUs, 16 GB of RAM, and 100 GB of local disk. The cluster was in the v1.28.12 version of Kubernetes, used CoreDNS as the cluster's local DNS server, and had Cilium as its CNI (Container Networking Interface). Cilium was installed using the official Cilium Helm Chart, with both the Helm Chart and the Cilium versions at version 1.15.2.

Additionally, some tooling was installed in the Kubernetes cluster: a Prometheus[56]

instance to store cluster and application monitoring data and a Grafana[47] instance to visualize Prometheus data.

4.2 Test environment: applications

To carry out tests and explorations, a test environment with real applications or applications that simulate real behaviors is needed, together with some way of generating and managing traffic that simulates the behavior of real users.

DeathStarBench [28], as aforementioned, is widely recognized in the field of microservices research and was used on Sinan[80]. It consists of a collection of representative microservices workloads that serve as a basis for evaluating general system aspects. Even so, after several experiments, it was decided not to use DeathStarBench due to a few reasons: (1) immaturity in parts of the deployment infrastructure, such as the lack of autoscaling for the suite when deployed in Kubernetes and lack of application resources requests and limits specification; (2) recurring problems in the documentation and with small bugs; (3) use of niche communication protocols, such as Apache Thrift using RPC Thrift Binary protocol, which is widely adopted but still much less used than HTTP; (4) problems with load testing mechanisms; (5) use of NoSQL databases, which are also widely used but still much less than conventional SQL databases. For some of these issues, such as 1 and 2, contributions were submitted and approved in the code repository[19], but it was decided that the effort put into resolving the other points would not be worthwhile. One of the DeathStarBench applications, specifically the e-commerce application, uses REST over HTTP and could be more easily used, but it is not yet widely available, and contacts asking for early access have not received any feedback.

The chosen suite was Train Ticket[27], a microservices-based train ticket booking system with 41 microservices, maintained by *Software Engineering Laboratory of Fudan University* and used in various microservices research. Unlike DeathStarBench apps, Train Ticket uses HTTP REST as the protocol for communication between the services and uses only SQL databases (MySQL). The deployment of Train Ticket was done using the official Kubernetes manifests, which also brings additional deployment options, allowing the deployment of the services with independent databases, and integration with one Apache SkyWalking[25] instance for tracing and monitoring.

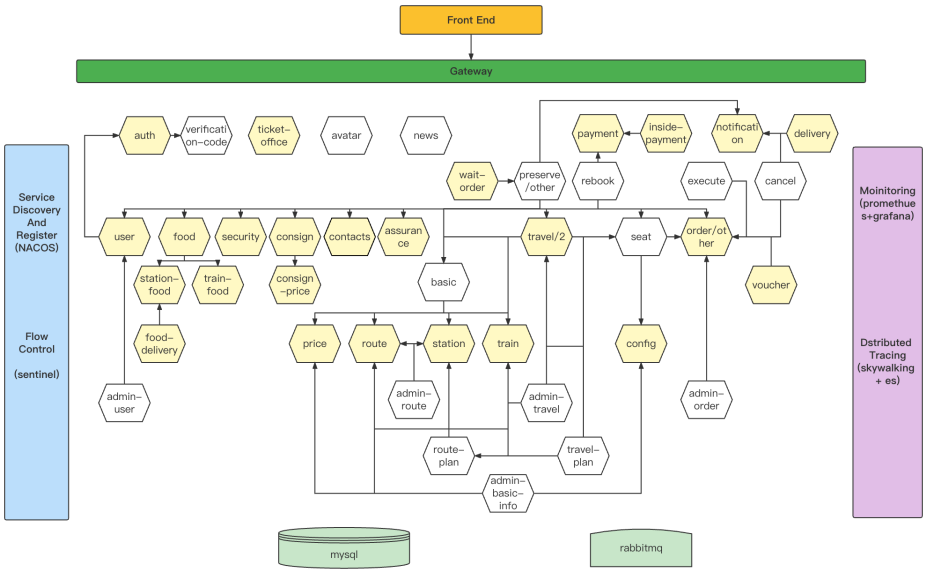


Figure 12: Fudan SE lab's Train Ticket Architecture. Image from official Github repository.

Train Ticket is a complex system – with 41 microservices –, and has a system called "`ts-gateway-service`" as the gateway to all of them. The architecture of the Train Ticket system is presented in Figure 12. To test flows involving all services, the authors made some request flows available, in Python code, which tests the systems by performing actions that real users would do. Still, these flows are not adapted to be run in load/stress tests. For this, there were some code snippets in a non-official third codebase[15], that use the Locust tool to implement flows inspired by the train ticket request flows repository.

Locust is an open-source load-testing tool designed to assess the performance of websites and applications. It allows the simulation of a high number of concurrent users interacting with a system to identify potential bottlenecks and measure response times under stress, generating reports of errors, systems response time, and ability to respond to requests.

The mentioned designed Locust test scenarios had not been maintained for a long time and contained various bugs. By going deeper into details and fixing them, the tests worked as expected – and the fixed tests will be published to a new fork of the non-mantained repository for future interested parties.

Locust uses a concept of "test scenario", where each scenario, implemented by the tester, is executed against the real environment. The scenarios we used for the tests are better described in the following sections.

4.3 Hubble Flows collection: Layer 4 proofs of concept

For the proof of concept of the Layer 4 flows collection, we used the Hubble metrics methodology to collect the flow data of the Train Ticket system, while running a load test to generate enough metrics for testing.

Our main objective was mapping the number of calls exchanged by the systems, to later calculate the degree of dependencies between them. Since Train Ticket applications were configured to communicate using HTTP 1.1 without `Keep-Alive` configuration, it is correct to assume that with Layer 4 information, we can determine the number of connections (requests) between applications by counting the occurrences of TCP SYN flags.

For configuring the Hubble metrics, Hubble accepts one string with the desired metrics and labels – the earlier mentioned "Context Options". For the tests, the Hubble Metrics were configured using the string presented in Listing 1. The referred Hubble metrics configuration enables detailed metrics collection for HTTP, Hubble Flows, and TCP traffic, with specific labels for enhanced context and traceability: source and destination IPs, namespaces, and workloads, along with traffic direction. It also enables DNS requests and Cilium drop metrics.

Listing 1: Hubble Metrics configuration

```
{
    httpV2:exemplars=true;labelsContext=source_ip\,source_namespace\,source_workload\,
        destination_ip\,destination_namespace\,destination_workload\,traffic_direction,
flow:labelsContext=source_ip\,source_namespace\,source_workload\,destination_ip\,
    destination_namespace\,destination_workload\,traffic_direction,
tcp:labelsContext=source_ip\,source_namespace\,source_workload\,destination_ip\,
    destination_namespace\,destination_workload\,traffic_direction,dns,drop
}
```

During our tests, however, an exacerbated value in the dropped flows metrics was noticed – the `hubble_lost_events_total` metric, which is enabled through the *drop* option –, with the label $source = hubble\_ring\_buffer$. This metric indicates the number of events that Hubble was unable to process from the specified source – without affecting the workloads connection. Therefore, the conclusion was that some events from the Hubble Ring Buffer were not being processed at the desired time.

The community has had some discussions about these points, and understanding the path of the data flows is crucial to solving the problem. The data of processed packets/flows

eBPF Map is processed by the Cilium userspace agent and sent to the Hubble Event Queue, and during this process, the agent can be overloaded and lose some events - the source label for the lost events metric in this case has the value $perf\_event\_ring\_buffer$. After that, the Event Queue data is read by the Hubble Observer to be sent to the ring buffer; if the Hubble Observer reads messages slower than the Event Queue is being populated, then the Events Queue can get full, and events will also be dropped, with the source label set to *observer_events_queue*. Finally, the events in the Hubble Ring Buffer are read by some client – for example, via API –, and then it is possible that the number of events put into the ring buffer outpaces the client read rate - causing some events to be overwritten and consequently dropped with the source label $hubble\_ring\_buffer$. The last one was the experienced problem.

The events drop is fundamentally a scaling problem. Cilium support parameters for configuring Hubble Events Queue Size (parameter `hubble-event-queue-size`) and Hubble Ring Buffer Capacity (parameter `hubble-event-buffer-capacity`), and aligning them is necessary in case of the existence of lost events. The other event drop scenarios can be solved with Cilium's resources scaling.

Thus, to solve the problem in the test scenario, the value for the Event Buffer Capacity was set to 65535 since the experienced lost metrics contained the `source=hubble_ring_buffer` label. During the test, problems with Hubble (Observer) Queue Size were not experienced, but it is important to note that it is automatically calculated based on the runtime number of CPUs and can be set manually if necessary.

Cilium also introduces a "monitor aggregation" parameter, which enables the coalescing of tracing events, to "only include periodic updates from active flows, or any packets that involve an L4 connection state change". The valid values are none, low, medium, and maximum. The "none" configuration causes a tracing event to be generated for every packet received and sent. Setting "low" causes an event to be generated for every packet sent. The "medium" option, which is the default one, generates events for every new connection, any time a packet contains TCP flags never before seen for the direction of packets in the packet, and on average once per `monitor-aggregation-interval` if a packet is seen in the interval; within "medium", TCP flags and reporting intervals are tracked independently in each direction, and an event

is generated for each packet that Cilium would have dropped. The “maximum” option is just an alias for the most aggressive aggregation level, which is currently the ”medium” one. This feature provides adjustable levels of detail in tracing events to monitor specific needs.

As the objective at the Layer 4 level is to count the TCP connections exchanged between systems and use it to map the dependency between them, the “low” option is the one that optimizes the necessary data with the best grouping. Using the “medium” option would imply in having distorted values, since if for instance 5 packets are seen at the same monitor-aggregation-interval, they would be grouped and counted as only one in the metric. Having these clarifications and after testing them in real environments, the environment was configured using the “low” option.

For testing, it was also necessary to configure the Train Ticket’s application resources. The resources of all applications were statically set to 1000m of CPU and 700Mi of memory, with no auto-scaling configured yet. Even though the official Train Ticket repository does not define ideal values for applications resources, these values were estimates found by observing the behavior of applications during preliminary tests. It is also important to highlight that even though these are not the ideal values, and that the ideal value of each application probably differs from the others due to their profiles, the designated values were sufficient for all requests to be responded to successfully.

With the environment fully configured, the workload generator was successfully executed using test scenarios defined via the Locust tool. While the specific details of these scenarios are not pertinent to this section and will be examined in subsequent experiments, their primary role here is to ensure that requests are generated between applications. The focus at this stage is solely on having inter-service communication and validating it through the generated metrics.

During the tests, it was possible to open the Hubble UI and see the application’s connections that occurred during the actions, as shown in Figure 13. Hubble also presented layer 4 information about the packets. Even so, Hubble does not present any kind of requests per second information or ratio of communication between the services.

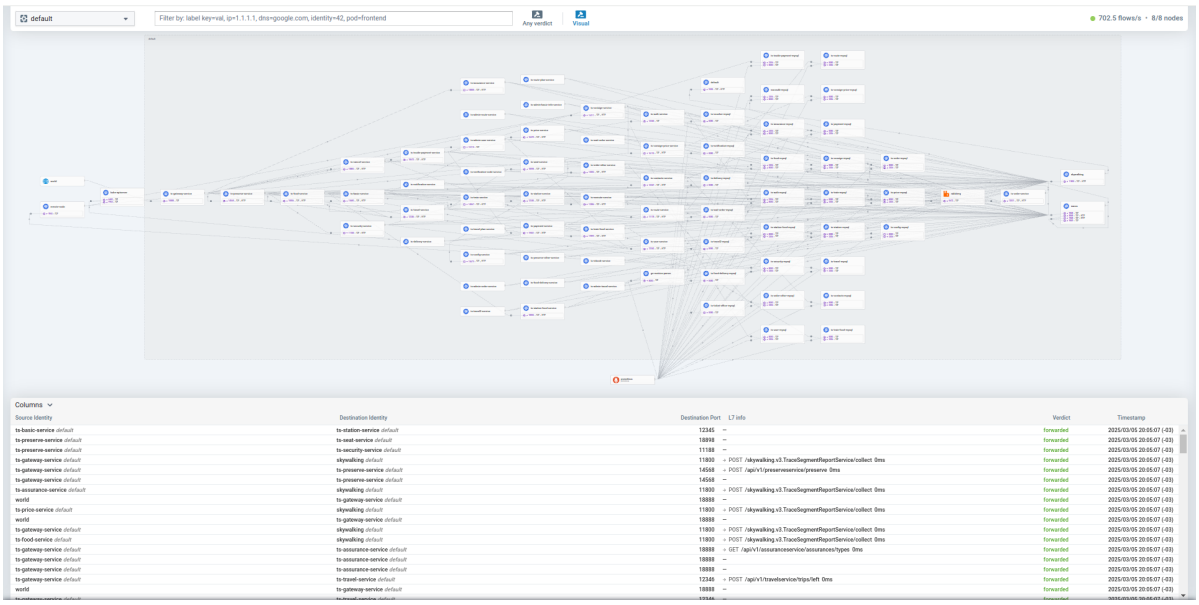


Figure 13: Train Ticket applications connections in Hubble UI

Together with visualizing the flows in the UI, metrics are generated by the Cilium Agents and exposed through the metrics endpoint. Examples of these metrics values are presented in Listing 2. The first metric indicates 18 TCP packages with the "SYN" flag from `ts-gateway-service` to `ts-inside-payment-service`, and the second one indicates 494 flows between the same services. As explained earlier, depending on the `monitor-aggregation-level` configuration, the second metric could also indicate the number of TCP packets.

Listing 2: Cilium Agent's Hubble metrics

```
1 hubble_tcp_flags_total{destination_ip=``192.168.3.238",destination_namespace=``default",
    destination_workload=``ts-inside-payment-service",family=``IPv4",flag=``SYN",source_ip
    =``192.168.3.160",source_namespace=``default",source_workload=``ts-gateway-service",
    traffic_direction=``egress"} 18
2 hubble_flows_processed_total{destination_ip=``192.168.3.238",destination_namespace=``default
    ",destination_workload=``ts-inside-payment-service",protocol=``TCP",source_ip
    =``192.168.3.160",source_namespace=``default",source_workload=``ts-gateway-service",
    subtype=``to-proxy",traffic_direction=``egress",type=``Trace",verdict=``FORWARDED"} 494
```

The metrics, however, are distributed among all the node agents, and the values shown in Listing 2 refer specifically to a single agent, which only captures data from the node on which it is running.

Moreover, although the example metrics in Listing 2 include the *source_workload* label,

there is an open upstream issue related to this label. Specifically, the label is absent in scenarios where the source and destination pods are located on different nodes. An example illustrating this issue is presented in Listing 3.

Listing 3: Cilium Agent's Hubble metrics missing source workload example

```
hubble_flows_processed_total{destination_ip="192.168.3.238",destination_namespace="default",
    destination_workload="\texttt{ts-inside-payment-service}",protocol="TCP",source_ip
    ="192.168.12.162",source_namespace="default",source_workload="",subtype="to-endpoint",
    traffic_direction="egress",type="Trace",verdict="FORWARDED"} 27118
```

The community is actively discussing a fix for the issue involving missing data; however, since it is possible to identify a pod by its IP address, the destination workload can still be inferred by locating the pod that matches the value of the *source_ip* label. To address this limitation, a system named `metrics-parser-api` was developed and deployed within the cluster. This system exposes a single API endpoint: `GET /metrics`. When invoked, the API scrapes all Cilium Node Agents for the specified metrics, resolves the destination workload using the destination IP, and returns a unified view of the metrics across all agents – ensuring the destination workload field is correctly populated whenever possible. To distinguish these enhanced metrics from the original Cilium metrics, new metric names were used, such as the earlier mentioned `hubble_parsed_tcp_flags_total` and `http_parsed_requests_total`. The code of the `metrics-parser-api` application is presented in Appendix D.

The cluster's Prometheus instance was configured to scrape and save data from the `metrics-parser-api` every 30 seconds. After testing, the setup enabled the following query in Prometheus:

```
sum by (source_workload, destination_workload) (increase(
    hubble_parsed_tcp_flags_total{traffic_direction="egress",flag="SYN
    "}[5m]))
```

Which counts the number of TCP SYN flags exchanged between systems in the last 5 minutes, originating from the *source_workload* (using $traffic\_direction$) and arriving at the *destination_workload*.

To ensure the accuracy of the metrics, the logs of specific services were inspected and counted against the metrics. This verification confirmed that the recorded metric values

aligned with the actual requests exchanged between services. Subsequently, Layer 7 metrics were also used to cross-validate the data obtained from the Layer 4 metrics, reinforcing the consistency and reliability of the measurements.

In conclusion, while certain specific configurations are required, collecting inter-service communication data at Layer 4 using Hubble Metrics is entirely feasible. Just as metrics can be obtained directly from the node agents, the same data can also be accessed through alternative approaches previously discussed – such as subscribing to the Hubble Ring Buffer for real-time flow parsing, or reading historical flow data exported by the Hubble Exporter.

Finally, it is fundamental to highlight that using HTTP 1.1 without Keep-Alive, as seen in Train Ticket, is quite common in microservices and can be generalized to a wide spectrum of applications. However, some applications use other technologies, such as Keep-Alive enabled, HTTP 2, or RPCs using pure TCP connections. In these cases, it is necessary to find alternative methods to measure the number of calls between systems, such as using information at the Layer 7 level or seeking other generalizations at the Transport Layer (Layer 4), such as counting TCP packets using the `monitor-aggregation-level=low` parameter.

4.4 Hubble Flows collection: Layer 7 proofs of concept

The process for collecting flows to map communication between services at the Layer 7 level is very similar to that of Layer 4. Fundamentally, it is a collection of metrics such as in the case of Layer 4. However, Cilium's internal process for generating these metrics has an additional step in the case of Layer 7 metrics.

As previously mentioned, to check the Application Layer (Layer 7) information level, Cilium uses an Envoy Proxy[5] to inspect, filter, and manipulate application-layer traffic, enabling deep visibility, fine-grained security, and advanced traffic management capabilities. The Envoy Proxy can be deployed together with Cilium Agents as a separate process within the Cilium Agent pod, or as independently life-cycled *DaemonSet*. Communication between the Cilium Agent and the Envoy Proxy takes place in both deployment modes via UNIX domain socket.

The use of a separate Envoy to the Cilium Agent deployment, however, brings some advantages: Cilium Agent can be restarted without affecting the live traffic proxied via

Envoy; Envoy can be patched (e.g. release upgrades) without affecting the Cilium Agent; it enables different CPU and memory configurations for the components and dedicated health probes for the Envoy proxy; and the logs are separated for clear troubleshooting. Still, for the aim of this research, none of these advantages would bring us significant benefit, and then the Envoy in the embedded mode was used, together with Cilium Agent, as it has the simplest installation.

While the separation of Cilium and Envoy for Layer 7 data seems odd, there are several reasons why it makes sense to keep Layer 7 introspection within Envoy. Envoy is built from the ground up to deal with Layer 7 traffic and is exceptional in it. It is also an excellent choice for tasks such as HTTP/2 and gRPC proxying, TLS termination, and high-level load balancing. Moreover, it already offers a variety of functions necessary for Layer 7 processing, such as intelligent routing, rate limiting, retries, circuit breaking, and metrics. Finally, the nature of the filter architecture means it can be extended with custom, unique filters that don't require changing the Cilium core codebase. All of this ensures that Cilium is as clean as possible, ensuring that both parts are easy to maintain and evolve independently of each other.

The Cilium's Envoy pipeline is also very flexible. It contains implemented protocols such as HTTP, but it can also be extended by implementing new parsers via the Go Extensions *proxylib* framework, allowing even not well-known protocols to be incorporated into the pipeline.

Although it is debatable that this additional layer of Envoy can cause performance problems, the distribution of Envoys is different from what is seen in Service Meshes. In the case of Envoy Proxy, we have one Envoy Proxy per cluster node, instead of one per application as for Service Meshes. Even so, performance problems may appear, but they can be overcome as they are an application scaling problem – in this case, the Envoy Proxy scaling if it is dedicated, or the Cilium Agent if it is embedded.

Given the context, the metrics configurations already made in the previous section, specifically the *httpV2* part, are sufficient to have HTTP metrics in Hubble. The two available HTTP metrics are: `http_requests_total`, and `http_request_duration_seconds`, and the first was used for the mapping. Even so, it also experiences the issue of the lack of in-

formation presented for the TCP metrics, and the same strategy of using an intermediary `metrics-parser-api` was adopted to solve it.

After performing the same tests used for the TCP metrics, an equivalent Prometheus query can be executed to retrieve HTTP communications between services. The following query counts the number of HTTP requests exchanged in the last 5 minutes, originating from the *source_workload* (based on the $traffic\_direction$) and targeting the *destination_workload*.

```
sum by (source_workload, destination_workload) (increase(
   hubble_parsed_http_requests_total{traffic_direction="egress"}[5m]))
```

4.5 Dependency graph

Given the notations and the metrics, it is possible to implement the calculation of the dependency degrees. This process is divided into two phases. First, the construction of a dependency graph, where each application is represented as a vertex, and the edges represent the number of requests between the connected applications. Later, the graph is transformed so that the edges reflect the values of the degree of dependencies between the connected applications.

It is important to highlight that this step is independent of the origin of the data source, which may be data collected via calculation through Layer 4 metrics, Layer 7 metrics, or provided from other mechanisms.

For the calculations, the `dependency-calculator` application is introduced, and will be responsible for computing the dependency and dependency degree graphs. The code of the `dependency-calculator` is presented in Appendix C. For the calculation, the data source will be the Prometheus, which will get data by scraping the `metrics-parser-api` application.

By executing the Prometheus query

```
sum by (source_workload, destination_workload) (increase(
   hubble_parsed_http_requests_total{traffic_direction="egress"}[10m])
   )
```

it is possible to retrieve the number of requests between all applications in the last 10 minutes using Layer 7 metrics, i.e. $\beta(X_i, X_u, 10)$. By iterating through the query response it is possible

to create the graph so that if $\beta(X_i, X_u, t) = Z$, then there is an edge with value Z between vertices X_i and X_u. The same can be achieved with Layer 4 metrics with another query:

```
sum by (source_workload, destination_workload) (increase(
    hubble_parsed_tcp_flags_total{traffic_direction="egress",flag="SYN
    "}[10m]))
```

Figure 14 exemplifies the graph built for the Train Ticket suite deployed in our test cluster after some minutes of Locust testing. The graph was computed through `dependency-calculator` application, and the image was generated using the `Graphviz` software. Figure 14 is very large and therefore difficult to visualize, so Figure 15 provides a crop of the image for better viewing.

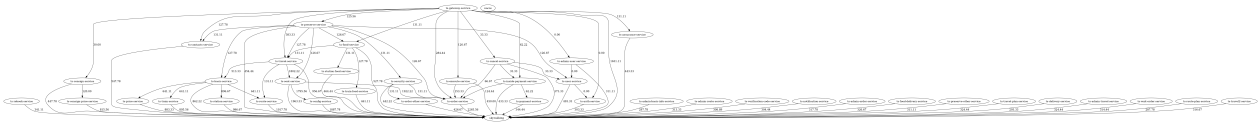


Figure 14: Train Ticket applications dependencies

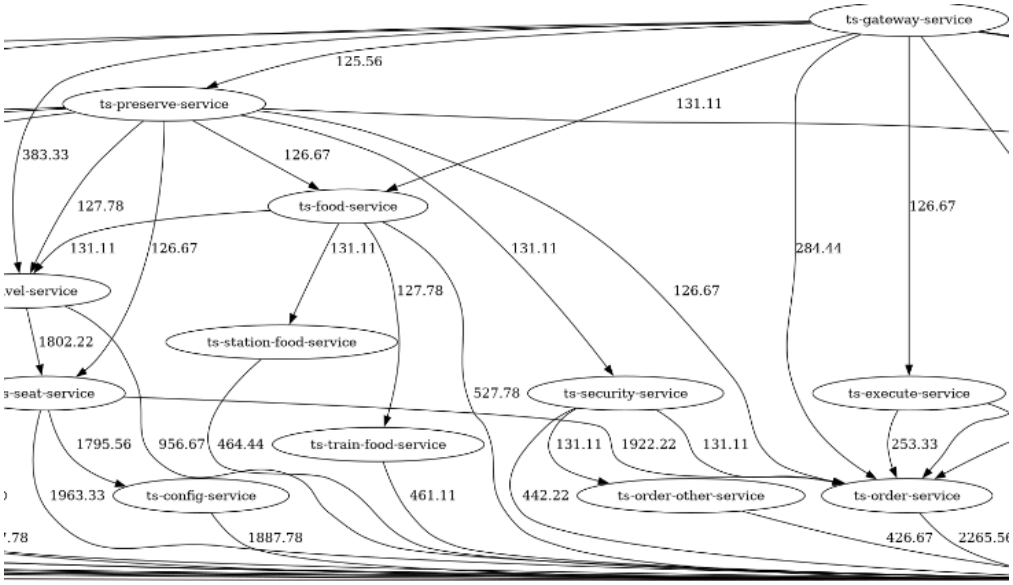


Figure 15: Zoom in on a section of the Train Ticket applications dependencies graph

4.6 Dependency degrees graph

Having the dependency graph, which contains the number of requests between two applications, the only data missing to applying the formula earlier stated is the number of received requests for each application, called $\alpha(X_i, t)$.

$\alpha(X_i, 10)$ can also be retrieved using a Prometheus query against `hubble_parsed_http_requests_total` or `hubble_parsed_tcp_flags_total`:

```
# Layer 4 metric
sum by (destination_workload) (increase(hubble_parsed_http_requests_total{traffic_direction="egress
    "}[10m]))
# Layer 7 metric
sum by (destination_workload) (increase(hubble_parsed_tcp_flags_total{traffic_direction="egress",flag:"
    SYN"}[10m]))
```

With the dependency graph and the received requests information, it is possible to calculate the dependency degree graph. The algorithm for calculating it consists of iterating through all pairs of applications (X_i, X_u) in the system and determining the degree of dependency between them. As stated before, the dependency matrix is computed using a recursive function $S(X_i, X_u, d, t)$, which considers both direct communications (when an edge exists between nodes in the graph) and indirect dependencies by traversing intermediary applications. The recursion depth is controlled by the parameter d, ensuring that dependencies beyond a certain level are not considered. Additionally, the shortest path function is employed to determine the minimal number of hops between services, filtering out dependencies that exceed the defined depth. The complete implementation of the algorithm is presented in Listing 4.

Listing 4: Dependency degree calculation

```
1  // Consider
2  // * alpha  = ReceivedRequests
3  // * beta   = SentRequests
4  // * mu     = DependencyDepth
5
6  Function CalculateDependencyDegreeGraph(Graph, ReceivedRequests, N, d, t):
7      // Create an empty matrix to store the dependencies
8      DependencyMatrix = CreateMatrix(N, N)
9
10     // For each pair of applications (X_i, X_u) in the graph, calculate their dependence
11     for i from 0 to N-1:
12         for u from 0 to N-1:
13             // Skip self-dependency (X_i cannot depend on itself)
14             if i == u:
15                 DependencyMatrix[i][u] = 0
16             else:
```

```
                // Calculate the degree of dependence S(X_i, X_u, d, t)
                DependencyMatrix[i][u] = S(Graph, ReceivedRequests, i, u, d, t)

    return DependencyMatrix


Function S(Graph, ReceivedRequests, X_i, X_u, d, t):
    // Base case: Check if X_i directly communicates with X_u
    if Graph[X_i][X_u] > 0:  // There is a direct communication
        return Graph[X_i][X_u] / ReceivedRequests[X_i]

    // Base case: If d is less than the dependency depth, return 0
    if d < DependencyDepth(Graph, X_i, X_u):
        return 0

    // Recursive case: Calculate the degree of dependence for indirect communication
    total_dependence = 0
    for each path P from X_i to X_u through intermediate applications X_w:
        path_dependence = 1
        for each intermediate application X_w in P:
            path_dependence *= S(Graph, ReceivedRequests, X_i, X_w, d-1, t) * S(Graph, ReceivedRequests,
                X_w, X_u, d-1, t)
        total_dependence += path_dependence

    return total_dependence


Function DependencyDepth(Graph, X_i, X_u):
    // Calculate the minimum number of hops from X_i to X_u in the dependency graph
    // This can be done using a shortest path algorithm (e.g., BFS or DFS)
    return ShortestPath(Graph, X_i, X_u)


Function ShortestPath(Graph, X_i, X_u):
    // Find the shortest path (minimum number of hops) from X_i to X_u using BFS or DFS
    // Return the number of hops
    visited = [False] * len(Graph)
    queue = [(X_i, 0)]  // Queue holds pairs of (node, depth)

    while queue:
        current_node, depth = queue.pop(0)
        if current_node == X_u:
            return depth
        if not visited[current_node]:
            visited[current_node] = True
            for neighbor in range(len(Graph)):
                if Graph[current_node][neighbor] > 0 and not visited[neighbor]:
                    queue.append((neighbor, depth + 1))

    return float('inf')  // If no path is found, return infinity (no dependency)
```

Figure 16 and Figure 17 exemplifies the algorithm being applied to a subset of the Train Ticket suite deployed in our test cluster after running some test cases. The graphs were computed through `dependency-calculator` application, using Layer 4 and Layer 7 metrics, respectively, with depth d as 3, and the image was generated using the *Graphviz* software.

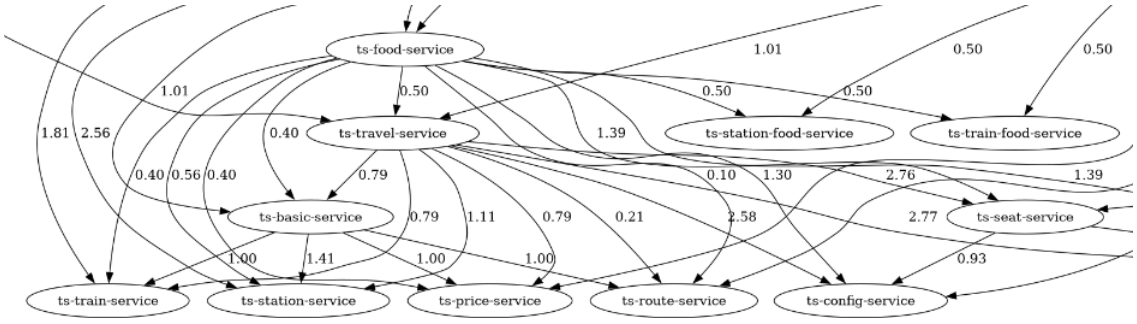


Figure 16: Section of Train Ticket dependency degree graph example using Layer 4 metrics

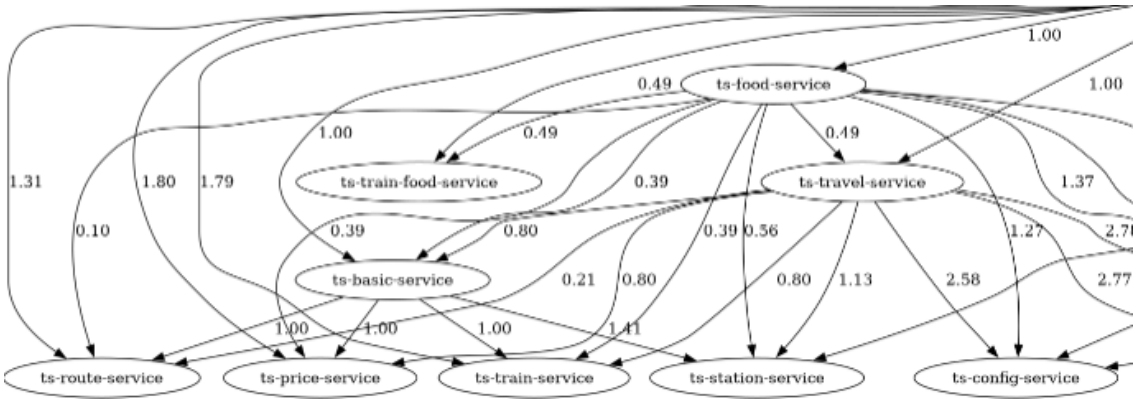


Figure 17: Section of Train Ticket dependency degree graph example using Layer 7 metrics

4.7 Dependency degree validation

To assess whether the dependency graph is a suitable method for understanding communication patterns among services and the potential effects of each dependency on its dependents, test scenarios from the Train Ticket Suite were employed. These scenarios were used to evaluate the cascading impacts that dependencies can produce within the system.

The experiments were carried out using the Locust tool, executed from within the cluster to reduce the influence of external factors on latency. Before each test, all databases were reset to ensure consistency. Each application was initially allocated with 100m of CPU and 100Mi of memory, with the ability to scale vertically up to 1000m of CPU and 700Mi of memory. It was confirmed that sufficient resources were available for all applications to reach their maximum limits, and that they were in the same initial state at the beginning of each test round.

The aim of these tests is not to definitively prove the accuracy of the generated dependency graph, but to demonstrate that the automatic mapping can support the understanding of how service dependencies influence system behavior. Considering this, the tests were designed to simulate overload scenarios in individual services and observe how these affect the performance of other services that depend on them, either directly or indirectly.

To simulate overloads, artificial latency was introduced in services that lie on critical paths within the dependency graph – particularly those that depend on or are depended upon by multiple other services. Latency was injected using the ChaosMesh[4] tool and was validated prior to the start of each test.

Additionally, some services natively have higher response times, especially those that aggregate responses from several other services. These services typically showed latencies above two seconds under normal conditions. This behavior was expected and accepted for the purpose of the experiments, as the number of users in each test was carefully selected to place sufficient load on the system without causing a high rate of errors. Therefore, the presence of high latency in the absence of a significant number of errors was considered an acceptable baseline for testing.

Scenario *UserNoLogin*

Scenario *UserNoLogin* is the simplest scenario of the tests, it simulates a user performing search operations without logging in:

1. `search_departure`: Searches for trips from "Shang Hai" to "Su Zhou".
2. `search_return`: Searches for trips from "Su Zhou" to "Shang Hai".

With the previously mentioned Locust tool, we ran tests against the Train Ticket using the following command:

```
locust -f locust/test.py --loglevel DEBUG -r 0.2 -u 10 -t 5m \
    -H <DEPLOYMENT_URL> --autostart UserNoLogin
```

The command describes a test that starts with 1 user, growing to 10 users in a 0.2 users/second rate with run time limit set to 5 minutes. The strategy of slowly growing the number of users aims to avoid burst requests, allowing the system to warm up before receiving requests. The test progress can be seen in the Figure 18.

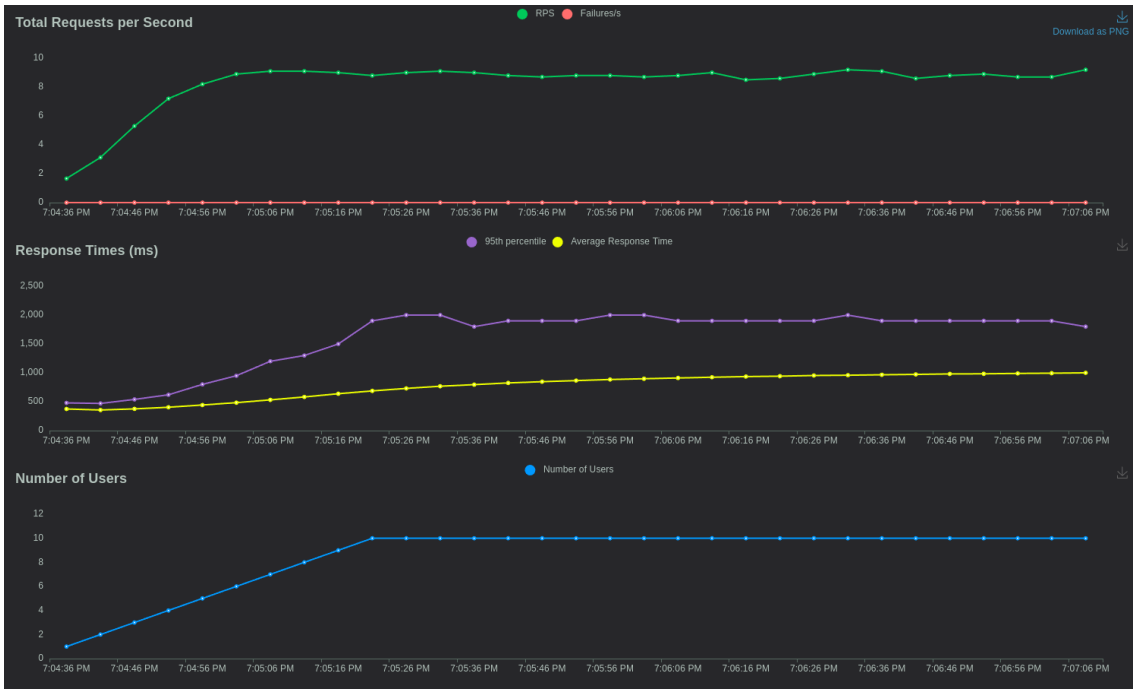


Figure 18: Locust test details for UserNoLogin experiment

The UserNoLogin scenario, as all other ones, starts with a call to `ts-gateway-service`, which forwards the call to the `ts-travel-service`, which is responsible for doing the trip searches of the flow using other services information. The dependency request chart and the dependency degree chart (with depths 1 and 2) of the `ts-travel-service` are presented, respectively, in the Figures 21, 19, and 20.

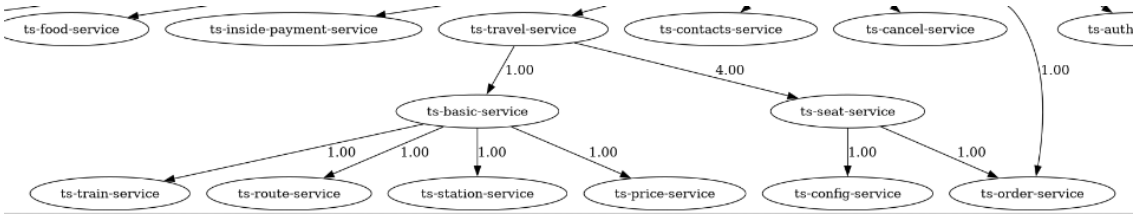


Figure 19: Dependency degree chart (depth=1) focused on the `ts-travel-service` for the UserNoLogin experiment

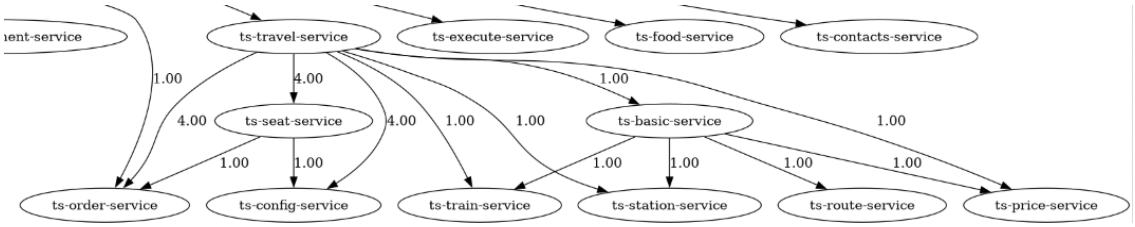


Figure 20: Dependency degree chart (depth=2) focused on the `ts-travel-service` for the UserNoLogin experiment

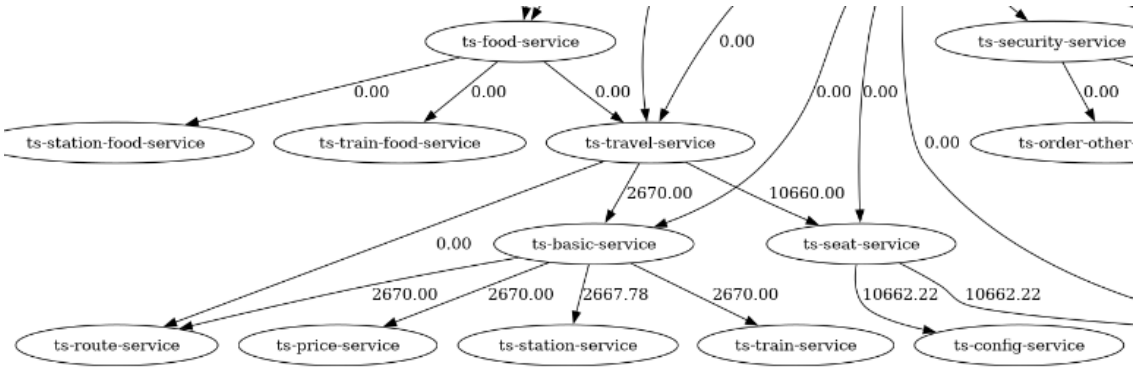


Figure 21: Dependency requests chart focused on the `ts-travel-service` for the UserNoLogin experiment

Looking at the charts, we can see that `ts-travel-service` is directly dependent on `ts-basic-service` and `ts-seat-service` and indirectly on `ts-train-service`, `ts-route-service`, `ts-station-service`, `ts-price-service`, `ts-config-service`, and `ts-order-service`. Each request to `ts-travel-service` triggers about 4 requests to `ts-seat-service` (along with its dependencies, `ts-order-service` and `ts-config-service`) and 1 request to `ts-basic-service` and its dependencies. Because of this, any delay in any dependency, such as `ts-seat-service`, will have a direct impact on `ts-travel-service`. For example, if `ts-seat-service` slows down by 100ms, the response time for `ts-travel-service` would increase by at least 400ms (since it makes 4 requests, each adding 100ms), and if `ts-route-service` slows down by 100ms, the response time for `ts-travel-service` would increase by at least 100ms.

Having the dependencies' impact clearly, the following tests were performed: added 50ms of additional latency to the `ts-seat-service`; added 100ms of additional latency to the

`ts-seat-service`.

Method	Requests	Failures	Min (ms)	Max (ms)	Median (ms)	Average (ms)	Current RPS	Failures/s
POST get_trip_information (+100ms `ts-seat-service`)	824	0	499.97	7046.57	2600.00	3050.67	2.76	0.0
POST get_trip_information (+50ms `ts-seat-service`)	1218	0	449.59	5565.38	1900.00	2060.65	4.08	0.0
POST get_trip_information	2271	0	255.27	2549.57	1000.00	1118.59	7.58	0.0

Table 1: Request statistics for *UserNoLogin* scenario with `ts-seat-service` increased latency

Method	25%	50%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%
POST get_trip_information (+100ms `ts-seat-service`)	1700	2600	4400	4600	5000	5300	5800	6100	7000	7000	7000
POST get_trip_information (+50ms `ts-seat-service`)	1200	1900	2900	3100	3400	3600	3800	4000	4400	5600	5600
POST get_trip_information	650	1000	1600	1600	1800	1900	2100	2200	2500	2500	2500

Table 2: Response time distribution for *UserNoLogin* scenario with `ts-seat-service` increased latency

With these tests, it is possible to verify that adding 100ms to the `ts-seat-service` causes the average latency of `ts-travel-service` requests to increase from about **1118.59 ms** (baseline) to **3050.67 ms**, reflecting an approximately 173% jump. This far exceeds a simple 4×100 ms addition because small increments of latency in a downstream service can magnify at runtime due to queuing, concurrency, and the chained nature of microservices. In other words, the extra overhead in `ts-seat-service` not only adds time to each request but also prolongs how long subsequent or parallel requests must wait, thereby inflating overall response times.

Because the test likely runs under a fixed duration or fixed concurrency model, each request taking longer means the system can complete fewer total requests within the same time window. Indeed, the total number of processed requests drops drastically as latency grows, from **2271** (no extra delay) to **1218** (+50 ms) and down to **824** (+100 ms). Likewise, both the median and average response times illustrate the extent of the performance hit. The median (50th percentile) climbs from **1000 ms** (baseline) to **1900 ms** with a 50 ms additional latency and **2600 ms** with 100 ms additional latency. The average rises from **1119 ms** to **2060.65 ms** (+50 ms) and **3050.67 ms** (+100 ms), confirming that what seems like a modest service delay can double or triple core latency metrics.

As requests spend more time "in flight", the system's throughput (requests per second) also falls: from **7.58 RPS** in the baseline to **4.08 RPS** (+50 ms) and **2.76 RPS** (+100 ms). Because concurrency limits quickly come into play, fewer requests get completed within any

given interval. Furthermore, the distribution of response times shifts significantly: the 25th percentile rises from **650 ms** at baseline to **1700 ms** with additional 100 ms delay, the 90th percentile from **1800 ms** to **5000 ms**, and tail latencies (e.g., 99.% or max) from **2500 ms** to **7000 ms**. These inflated high-percentile values reveal that even a small injected delay can push some requests into multi-second wait times, providing a stark illustration of the importance of tail-latency metrics.

From a microservices theory perspective, these results demonstrate how service interdependencies drive latency far beyond simple additive effects. In this scenario, `ts-travel-service` depends on `ts-seat-service`, so adding a minor delay to `ts-seat-service` can create significant performance degradations for `ts-travel-service`, especially under load. Sequential or parallel calls to `ts-seat-service` often compound this overhead, and concurrency further amplifies it. The increased latency also drives down the system's throughput because each request lingers longer in processing pipelines before freeing resources for the next request. This underscores why designing inter-service communication patterns—through caching, asynchronous/event-driven architectures, load balancing, or other optimizations—is essential to prevent a single slowdown from escalating into a full bottleneck.

The straightforward theory is that the greater the degree of dependence of application A on application B, the greater the impact of problems on application A on application B. Therefore, adding latency to `ts-basic-service` or one of its dependencies should affect `ts-travel-service` negatively, but less than adding latency to `ts-seat-service` or its dependencies, since `ts-travel-service` has dependency degree 4 with `ts-seat-service` and 1 with ts-basic service. Thus, the following tests were executed: added 100ms to `ts-price-service`; added 200ms to `ts-price-service`; added 400ms to `ts-price-service`.

Method	Requests	Failures	Min (ms)	Max (ms)	Median (ms)	Average (ms)	Current RPS	Failures/s
POST get_trip_information	2271	0	255.27	2549.57	1000.00	1118.59	7.58	0.0
POST get_trip_information (+100ms `ts-price-service`)	2253	0	355.15	2405.58	1100.00	1126.39	7.51	0.0
POST get_trip_information (+200ms `ts-price-service`)	1698	0	412.23	4406.43	1500.00	1489.12	5.67	0.0
POST get_trip_information (+400ms `ts-price-service`)	484	0	4534.42	5340.65	4900.00	5035.32	1.65	0.0

Table 3: Request statistics for *UserNoLogin* scenario with `ts-price-service` increased latency

The results are presented in Tables 3 and 4 and these additional tests reinforce the principle that the degree of dependence between services directly shapes the performance impact.

Method	25%	50%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%
POST get_trip_information	650	1000	1600	1600	1800	1900	2100	2200	2500	2500	2500
POST get_trip_information (+100ms `ts-price-service`)	740	1100	1500	1600	1700	1800	2000	2100	2300	2400	2400
POST get_trip_information (+200ms `ts-price-service`)	1300	1500	1700	1700	2000	2200	2600	2800	4000	4400	4400
POST get_trip_information (+400ms `ts-price-service`)	4800	4900	5200	5300	5300	5300	5300	5300	5300	5300	5300

Table 4: Response time distribution for *UserNoLogin* scenario with `ts-price-service` increased latency

When `ts-travel-service` relies heavily on a particular downstream service, even a small delay can ripple through the system. By contrast, when a service has a lower dependency degree, the effect of added latency can be comparatively mild—unless that latency becomes large. Specifically, `ts-seat-service` involves four separate calls from `ts-travel-service`, so **50–100 ms** added there previously caused significant spikes in response times and drops in throughput. However, `ts-price-service` is only called once per flow, so introducing a smaller **100 ms** overhead had a minor effect: total requests stayed near baseline (**2253** vs. **2271**), median latencies rose only slightly (**1000 ms** to **1100 ms**), and throughput remained essentially stable (**7.58** to **7.51 RPS**).

Once the artificial delay for `ts-price-service` increased to **200 ms**, the effect became more pronounced: median latency climbed to **1500 ms**, average to **1489 ms**, and throughput fell to **5.67 RPS**. With **400 ms** added, the impact was dramatic, pushing median latency to **4900 ms**, average latency to **5035 ms**, and throughput down to **1.65 RPS**. Notably, even at the **25th** percentile, requests were taking nearly five seconds, highlighting how large delays—even in a single low-degree dependency—can degrade the entire user-facing service. The key takeaway is that while a lesser dependency degree usually means lower sensitivity to modest latency, sufficiently large delays in any required call can still cripple performance through queueing, concurrency bottlenecks, and extended tail latencies.

Multiple scenarios: *UserBooking, UserConsignTicket and UserCancelNoRefund*

Even though the responsibility of each microservice in a system is small and well-defined, they are often used in more than one flow and for different calls. In the *UserNoLogin* scenario, we can observe that the lowest degree of dependency of the systems involved in the flow is 1, but in a real environment, because each system can be dealing with different flows and each flow has a different weight (that is, it is exercised disproportionately), we often see very

different degrees of dependency. In order to exercise a more complex flow, we will evaluate the tests running 3 scenarios in parallel:

Scenario *UserBooking*: simulates a user logging in, booking a trip, and completing the payment process:

1. `search_departure`: Searches for trips from "Shang Hai" to "Su Zhou".
2. `search_return`: Searches for trips from "Su Zhou" to "Shang Hai".
3. `book`: Books a trip.
4. `pay`: Pays for the booked trip.
5. `collect_and_use`: Collects the ticket and enters the station.

Scenario *UserConsignTicket*: simulates a user logging in, booking a trip, and consigning a ticket:

1. `search_departure`: Searches for trips from "Shang Hai" to "Su Zhou".
2. `search_return`: Searches for trips from "Su Zhou" to "Shang Hai".
3. `book`: Books a trip.
4. `consign`: Consigns the ticket.

Scenario *UserCancelNoRefund*: simulates a user logging in, booking a trip, and canceling it without a refund:

1. `search_departure`: Searches for trips from "Shang Hai" to "Su Zhou".
2. `search_return`: Searches for trips from "Su Zhou" to "Shang Hai".
3. `book`: Books a trip.
4. `cancel`: Cancels the booked trip.

For these scenarios, it is necessary to set the baselines. To do this, the following scenarios will be run: 12 users, each scenario with 33.33% of the requests; 21 users, each scenario with 33.33% of the requests; 51 users, each scenario with 33.33% of the requests.

The tests were run using the following command:

```
locust --equal-weights -f locust/test.py --loglevel DEBUG -r 1 -u
  <NUMBER_OF_USERS> --processes 3 -t 5m -H <DEPLOYMENT_URL> --autostart
      UserBooking UserConsignTicket UserCancelNoRefund
```

Which ensures that each scenario has 1/3 of the requests (`--equal-weights` parameter), and adds 1 user per second until the maximum limit for the test.

The results of these experiments are presented in Appendix A. An important conclusion is that adding more users did not increase the throughput of the tests. As previously stated, each application has predefined resource limits and no autoscaling configuration. Therefore, with 21 users, the system was already operating near the maximum service capacity of its applications. Increasing the number of users beyond this point did not translate into higher throughput but led to deterioration of the behavior of the system: the number of concurrent requests exceeded service handling capacity, forcing services to queue incoming requests while still processing the backlog, as previously explained. This behavior is clearly visible in Figures 22, 23, and 24, where the three test runs with 51 users showed inconsistent behavior and significantly increased latencies due to application overload.

In scenarios with 12 users, although the services were exercised, the system showed low resource consumption, low request failure rates (often zero, as seen in the "Failures" columns of Tables 9 and 12), and response times remained consistently low across all the flows. Additionally, the average response times and high percentiles (e.g., 90%, 95%) indicate minimal queuing and resource contention. This suggests that the system was operating below its capacity limits, leaving room for better CPU, memory, and concurrency utilization.

In contrast, with 51 users, the system showed clear signs of overload. This is evidenced by the increased number of request failures in operations such as `POST preserve_ticket` (e.g., 90 failures in the first run, 95 failures in the second, 101 failures in the third, as shown in Table 11). Furthermore, response times became highly variable, with the 99th and 100th percentiles reaching extreme values (e.g., 50 seconds for some flows), indicating that the system struggled to maintain service quality under this level of load.

Given these observations, 21 users were chosen as the test baseline. At this level, the system was neither underutilized nor severely overloaded. The response time distributions remained within acceptable limits, with minimal or no failures across all flows (as shown in Tables 10 and 13). Thus, 21 users allowed the environment to operate close to its optimal resource usage without triggering instability or significant queuing effects.

Figure 22: Multiple scenarios tests run 1 with 51 concurrent users

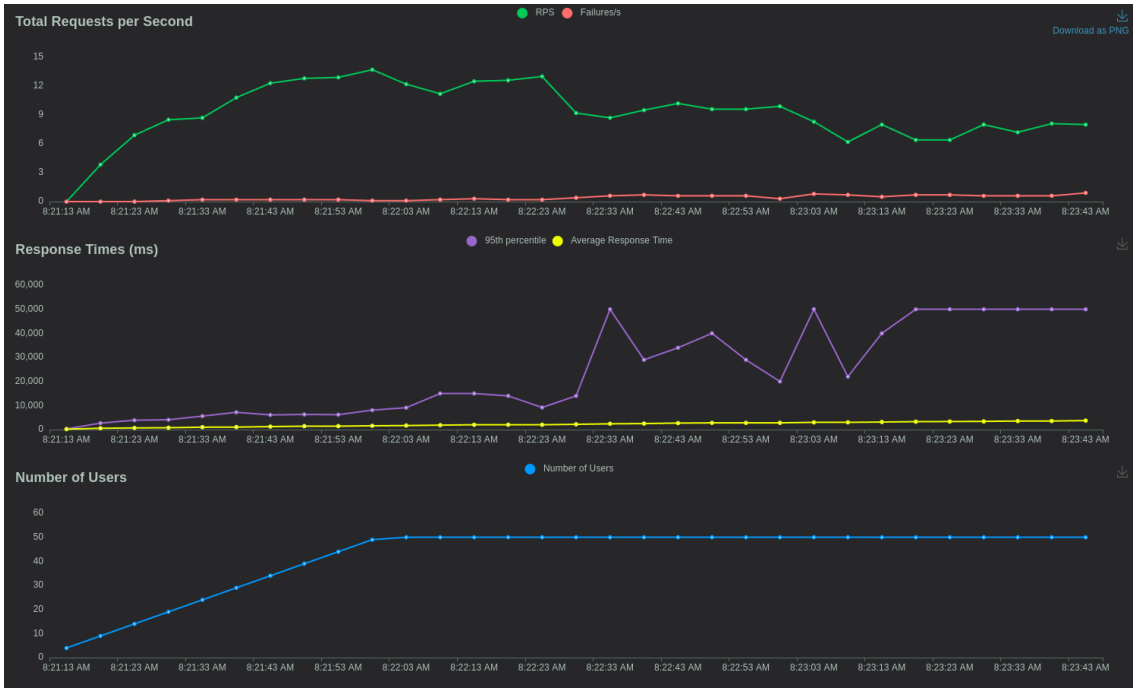


Figure 23: Multiple scenarios tests run 2 with 51 concurrent users

Figure 24: Multiple scenarios tests run 3 with 51 concurrent users

As in the other tests, it is also important to understand the applications connections for the current scenarios. The Figures 25, 26, and 27 presents, respectively, the requests and dependency graphs with dependency degree 1 and 2 during one of the test runs with 21 users.

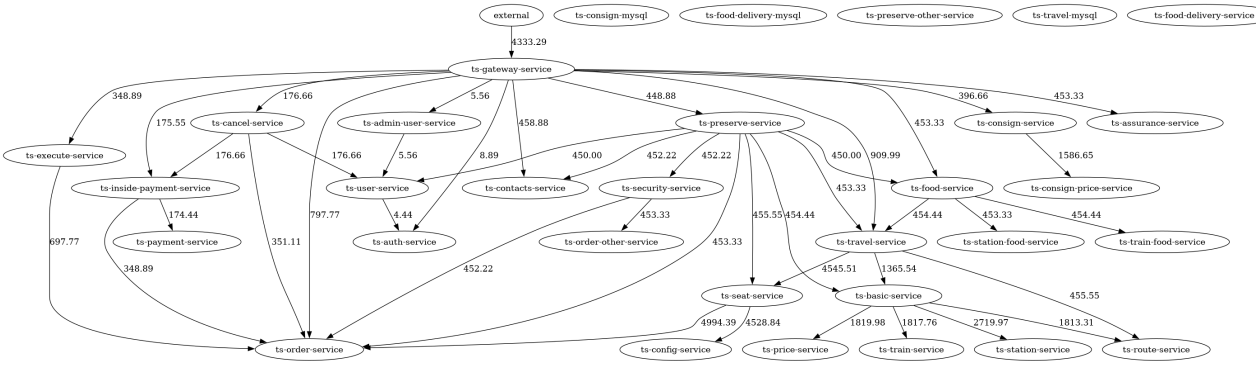


Figure 25: Requests graph of the multiple scenarios test with 21 users

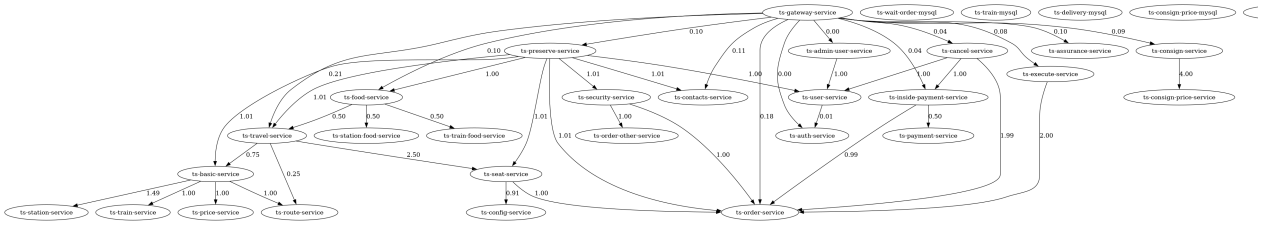


Figure 26: Dependency degree graph (with degree 1) of the multiple scenarios test with 21 users

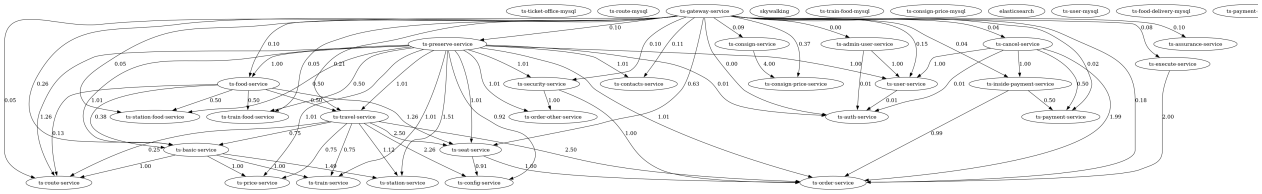


Figure 27: Dependency degree graph (with degree 2) of the multiple scenarios test with 21 users

Analyzing the dependency degree graph with degree 1, `ts-order-service` is one of the core applications of the environment, having a dependency for `ts-seat-service`, `ts-preserve-service`, `ts-security-service`, `ts-inside-payment-service`, `ts-cancel-service` and `ts-execute-service`. Thus, the tests focused on adding latency to it and seeing how it affects the multiple scenarios being tested. The following tests with 21 users, with traffic split equally between the scenarios, were executed: adding 100ms to `ts-order-service`; adding 200ms to `ts-order-service`; adding 400ms to `ts-order-service`.

As mentioned before, `ts-order-service` is one of the core services for the test's scenarios. Thus, any extra latency there can affect multiple endpoints at once. As seen in Table 5, even a 100 ms delay modestly reduces throughput (for instance, `GET cancel_order` requests drop from **160** down to **135**), while at 400 ms, the processed requests shrink dramatically (down to 41). This echoes previous tests, where artificially prolonged "in-flight" times limit how many new requests can be serviced concurrently.

No direct failures appeared in most endpoints except for `PUT create_consign`, which

Method	Requests	Failures	Min (ms)	Max (ms)	Median (ms)	Average (ms)	Current RPS	Failures/s
GET cancel_order	160	0	110.113	178.761	130.000	134.305	0.533	0.000
GET cancel_order (+100ms ts-order-service)	135	0	324.064	418.747	350.000	352.332	0.450	0.000
GET cancel_order (+200ms ts-order-service)	77	0	528.265	578.899	540.000	546.081	0.257	0.000
GET cancel_order (+400ms ts-order-service)	41	0	933.085	1006.360	950.000	950.817	0.137	0.000
GET collect_ticket	153	0	55.029	129.668	75.000	76.578	0.510	0.000
GET collect_ticket (+100ms ts-order-service)	125	0	279.297	346.005	290.000	296.634	0.417	0.000
GET collect_ticket (+200ms ts-order-service)	72	0	478.159	530.335	490.000	494.107	0.240	0.000
GET collect_ticket (+400ms ts-order-service)	38	0	885.204	935.699	900.000	897.394	0.127	0.000
PUT create_consign	131	131	879.156	1464.414	1100.000	1106.442	0.437	0.437
PUT create_consign (+100ms ts-order-service)	85	85	3853.670	4644.707	4200.000	4198.852	0.283	0.283
PUT create_consign (+200ms ts-order-service)	58	58	3937.008	4581.724	4100.000	4144.729	0.193	0.193
PUT create_consign (+400ms ts-order-service)	36	36	3675.562	4423.834	4100.000	4079.371	0.120	0.120
GET enter_station	153	0	56.517	137.124	76.000	79.181	0.510	0.000
GET enter_station (+100ms ts-order-service)	125	0	274.244	340.686	290.000	294.616	0.417	0.000
GET enter_station (+200ms ts-order-service)	72	0	480.315	527.855	490.000	496.719	0.240	0.000
GET enter_station (+400ms ts-order-service)	38	0	876.694	929.852	890.000	895.961	0.127	0.000
GET get_assurance_types	452	0	18.825	66.333	25.000	26.750	1.507	0.000
GET get_assurance_types (+100ms ts-order-service)	346	0	17.300	62.530	23.000	24.868	1.153	0.000
GET get_assurance_types (+200ms ts-order-service)	212	0	16.955	114.634	24.000	25.440	0.707	0.000
GET get_assurance_types (+400ms ts-order-service)	118	0	18.886	100.285	23.000	25.192	0.393	0.000
GET get_food_types	445	0	700.926	3855.800	1600.000	1681.772	1.484	0.000
GET get_food_types (+100ms ts-order-service)	344	0	94.473	2596.467	780.000	851.391	1.147	0.000
GET get_food_types (+200ms ts-order-service)	212	0	93.826	1395.065	200.000	381.156	0.707	0.000
GET get_food_types (+400ms ts-order-service)	117	0	96.137	1579.585	120.000	347.382	0.390	0.000
POST get_order_information	752	0	25.413	81.331	34.000	36.299	2.507	0.000
POST get_order_information (+100ms ts-order-service)	596	0	126.492	173.374	140.000	140.005	1.987	0.000
POST get_order_information (+200ms ts-order-service)	354	0	225.685	276.861	240.000	239.185	1.180	0.000
POST get_order_information (+400ms ts-order-service)	192	0	430.992	466.032	440.000	438.289	0.640	0.000
POST get_trip_information	889	0	213.634	736.353	380.000	377.322	2.964	0.000
POST get_trip_information (+100ms ts-order-service)	687	0	441.437	4334.140	1600.000	2027.544	2.290	0.000
POST get_trip_information (+200ms ts-order-service)	414	0	1323.932	8578.567	3500.000	4418.061	1.380	0.000
POST get_trip_information (+400ms ts-order-service)	225	0	2401.051	15866.503	6600.000	8560.682	0.750	0.000
POST pay_order	153	0	113.630	201.632	130.000	137.402	0.510	0.000
POST pay_order (+100ms ts-order-service)	125	0	331.696	435.741	350.000	356.677	0.417	0.000
POST pay_order (+200ms ts-order-service)	72	0	537.971	581.907	550.000	552.356	0.240	0.000
POST pay_order (+400ms ts-order-service)	38	0	931.102	1006.613	950.000	953.915	0.127	0.000
POST preserve_ticket	436	0	1805.176	42208.797	7800.000	9698.639	1.454	0.000
POST preserve_ticket (+100ms ts-order-service)	344	0	1894.787	5558.569	2900.000	3033.858	1.147	0.000
POST preserve_ticket (+200ms ts-order-service)	208	0	2842.567	7064.742	4400.000	4511.516	0.693	0.000
POST preserve_ticket (+400ms ts-order-service)	116	0	6235.441	11920.262	8300.000	8403.688	0.387	0.000
GET query_contacts	446	0	21.817	60.353	28.000	28.455	1.487	0.000
GET query_contacts (+100ms ts-order-service)	345	0	20.913	69.541	27.000	28.375	1.150	0.000
GET query_contacts (+200ms ts-order-service)	212	0	22.108	54.685	28.000	29.137	0.707	0.000
GET query_contacts (+400ms ts-order-service)	117	0	21.022	68.119	28.000	29.343	0.390	0.000

Table 5: Request statistics for multiple scenarios using 21 users with `ts-order-service` increased latency

consistently failed with a low rate regardless of delay, suggesting a small overload in all scenarios due to it complicated flow. Nevertheless, across endpoints such as `POST get_trip_information` and `POST preserve_ticket`, the added latency expanded not just the average response time but also the higher-percentile tails. For instance, `POST get_trip_information` median rose from **380 ms** to **6600 m**s at a 400 ms injection, and tail values (e.g., 99%) shot to **15000–16000 ms**. Similarly, `POST preserve_ticket` already had lengthy baseline requests –its 25th percentile was **4500 ms** – but rose to **7300 ms** under 400 ms latency, highlighting how extra overhead on a central service can multiply existing delays.

These new tests align with our previous findings on dependency degrees and concurrency saturation. A service like `ts-order-service`, essential to many steps across different user tasks, suffers a disproportionately large impact from even small latency increases.

Overall, the observations from injecting delay in `ts-order-service` reinforce the same

Method	25%	50%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%
GET cancel_order	130	130	140	140	150	160	170	170	180	180	180
GET cancel_order (+100ms `ts-order-service`)	340	350	360	360	370	380	400	410	420	420	420
GET cancel_order (+200ms `ts-order-service`)	540	540	550	550	560	570	580	580	580	580	580
GET cancel_order (+400ms `ts-order-service`)	940	950	950	960	970	970	1000	1000	1000	1000	1000
GET collect_ticket	70	75	80	82	89	95	110	120	130	130	130
GET collect_ticket (+100ms `ts-order-service`)	290	290	300	300	310	320	330	330	350	350	350
GET collect_ticket (+200ms `ts-order-service`)	490	490	500	500	500	510	510	530	530	530	530
GET collect_ticket (+400ms `ts-order-service`)	890	900	900	900	900	930	940	940	940	940	940
PUT create_consign	1000	1100	1200	1200	1200	1300	1300	1300	1500	1500	1500
PUT create_consign (+100ms `ts-order-service`)	4100	4200	4300	4300	4400	4500	4600	4600	4600	4600	4600
PUT create_consign (+200ms `ts-order-service`)	4000	4100	4200	4200	4300	4400	4500	4600	4600	4600	4600
PUT create_consign (+400ms `ts-order-service`)	4000	4100	4300	4300	4300	4400	4400	4400	4400	4400	4400
GET enter_station	71	76	83	86	99	110	120	140	140	140	140
GET enter_station (+100ms `ts-order-service`)	290	290	300	300	310	310	330	340	340	340	340
GET enter_station (+200ms `ts-order-service`)	490	490	500	500	520	520	520	530	530	530	530
GET enter_station (+400ms `ts-order-service`)	890	890	900	900	910	920	930	930	930	930	930
GET get_assurance_types	23	25	28	29	34	41	51	58	66	66	66
GET get_assurance_types (+100ms `ts-order-service`)	22	23	26	27	30	33	46	49	63	63	63
GET get_assurance_types (+200ms `ts-order-service`)	22	24	26	27	30	34	41	53	110	110	110
GET get_assurance_types (+400ms `ts-order-service`)	22	24	25	26	28	31	51	59	100	100	100
GET get_food_types	1400	1600	2000	2100	2300	2400	2500	2700	3900	3900	3900
GET get_food_types (+100ms `ts-order-service`)	370	790	1200	1300	1600	1800	2100	2400	2600	2600	2600
GET get_food_types (+200ms `ts-order-service`)	110	200	620	690	910	1200	1300	1300	1400	1400	1400
GET get_food_types (+400ms `ts-order-service`)	110	120	460	630	940	1300	1300	1400	1600	1600	1600
POST get_order_information	32	34	38	39	44	53	62	69	81	81	81
POST get_order_information (+100ms `ts-order-service`)	140	140	140	140	150	150	160	170	170	170	170
POST get_order_information (+200ms `ts-order-service`)	240	240	240	240	240	250	260	270	280	280	280
POST get_order_information (+400ms `ts-order-service`)	440	440	440	440	440	440	450	450	470	470	470
POST get_trip_information	300	380	430	440	490	540	600	640	740	740	740
POST get_trip_information (+100ms `ts-order-service`)	1100	1600	3000	3100	3400	3600	3800	3900	4300	4300	4300
POST get_trip_information (+200ms `ts-order-service`)	2400	3600	6500	6800	7100	7400	7600	7800	8600	8600	8600
POST get_trip_information (+400ms `ts-order-service`)	4500	6600	13000	13000	14000	14000	15000	15000	16000	16000	16000
POST pay_order	130	130	140	150	150	160	180	190	200	200	200
POST pay_order (+100ms `ts-order-service`)	350	350	360	360	370	380	390	390	440	440	440
POST pay_order (+200ms `ts-order-service`)	550	550	560	560	570	570	580	580	580	580	580
POST pay_order (+400ms `ts-order-service`)	950	950	960	960	970	980	1000	1000	1000	1000	1000
POST preserve_ticket	4500	7800	12000	14000	20000	24000	29000	34000	42000	42000	42000
POST preserve_ticket (+100ms `ts-order-service`)	2600	2900	3500	3600	4000	4200	4600	4800	5600	5600	5600
POST preserve_ticket (+200ms `ts-order-service`)	4000	4400	5000	5300	5700	5900	6300	6400	7100	7100	7100
POST preserve_ticket (+400ms `ts-order-service`)	7300	8400	9400	9500	10000	11000	11000	12000	12000	12000	12000
GET query_contacts	26	28	30	30	32	36	40	53	60	60	60
GET query_contacts (+100ms `ts-order-service`)	25	27	29	30	33	41	47	51	70	70	70
GET query_contacts (+200ms `ts-order-service`)	26	28	31	31	35	39	46	50	55	55	55
GET query_contacts (+400ms `ts-order-service`)	27	28	30	31	33	37	47	55	68	68	68

Table 6: Response time distribution for multiple scenarios using 21 users with `ts-order-service` increased latency

core principles seen with `ts-seat-service` and `ts-price-service`, also reiterating the importance of understanding service dependencies and concurrency thresholds. Under realistic multi-flow conditions, a central service's slowdown propagates widely, intensifying queue times and elevating high-percentile latencies. Each microservice has intrinsic resource limits, and once those are reached, performance stalls or worsens. Likewise, a frequently invoked service can degrade the entire system's behavior, especially under concurrency. In complex real-world environments, concurrency spikes can occur in unpredictable patterns, and certain flows may be exercised more heavily than others. Consequently, proactively identifying and alleviating bottlenecks in key services such as `ts-order-service` is critical for sustaining

robust performance. As a result, addressing latency in systems critical paths is essential to maintain overall application stability and responsiveness.

A final test was conducted to illustrate the opposite scenario of the `ts-order-service`. In this case, the dependency degree graph shows that the `ts-execute-service` is invoked exclusively by the `ts-gateway-service`, meaning that it participates in only one of the test flows and is not involved in any others. To assess its impact, we introduced latencies of 100 ms and 400 ms to the service and observed how this affected the other flows and the overall environment. The results are summarized in the Tables 7 and 8.

Method	Requests	Failures	Min (ms)	Max (ms)	Median (ms)	Average (ms)	Current RPS	Failures/s
GET cancel_order	167	0	107.24	360.77	140.00	144.00	0.56	0.00
GET cancel_order (+100ms `ts-execute-service`)	155	0	109.14	246.02	140.00	140.60	0.52	0.00
GET cancel_order (+400ms `ts-execute-service`)	176	0	107.13	316.24	140.00	139.86	0.59	0.00
GET collect_ticket	156	0	54.83	304.65	75.00	82.81	0.52	0.00
GET collect_ticket (+100ms `ts-execute-service`)	157	0	160.53	312.39	180.00	183.22	0.52	0.00
GET collect_ticket (+400ms `ts-execute-service`)	143	0	466.68	4090.72	480.00	515.54	0.48	0.00
PUT create_consign	107	107	331.85	8891.74	4300.00	4337.98	0.36	0.36
PUT create_consign (+100ms `ts-execute-service`)	113	113	263.26	5536.77	4400.00	4189.28	0.38	0.38
PUT create_consign (+400ms `ts-execute-service`)	113	113	371.74	8506.74	4400.00	4389.35	0.38	0.38
GET enter_station	156	0	55.68	1798.00	76.00	109.60	0.52	0.00
GET enter_station (+100ms `ts-execute-service`)	157	0	163.59	310.47	180.00	181.39	0.52	0.00
GET enter_station (+400ms `ts-execute-service`)	143	0	463.56	569.11	480.00	479.87	0.48	0.00
GET get_assurance_types	435	0	17.15	1760.31	24.00	30.41	1.45	0.00
GET get_assurance_types (+100ms `ts-execute-service`)	432	0	17.03	67.80	24.00	25.76	1.44	0.00
GET get_assurance_types (+400ms `ts-execute-service`)	436	0	15.37	1320.70	24.00	36.88	1.45	0.00
GET get_food_types	430	0	274.82	8809.00	1700.00	1832.97	1.43	0.00
GET get_food_types (+100ms `ts-execute-service`)	429	0	326.56	3779.67	1800.00	1928.77	1.43	0.00
GET get_food_types (+400ms `ts-execute-service`)	431	0	518.22	3724.51	1700.00	1776.51	1.44	0.00
POST get_order_information	744	0	23.71	4012.88	35.00	48.44	2.48	0.00
POST get_order_information (+100ms `ts-execute-service`)	741	0	25.37	373.56	35.00	37.79	2.47	0.00
POST get_order_information (+400ms `ts-execute-service`)	720	0	26.20	3861.82	35.00	73.40	2.40	0.00
POST get_trip_information	865	0	213.49	8564.84	410.00	432.99	2.88	0.00
POST get_trip_information (+100ms `ts-execute-service`)	857	0	198.21	891.92	390.00	376.62	2.86	0.00
POST get_trip_information (+400ms `ts-execute-service`)	870	1	213.67	50011.03	400.00	465.02	2.90	0.00
POST pay_order	156	0	111.60	575.06	140.00	148.87	0.52	0.00
POST pay_order 100ms	157	0	109.74	296.13	140.00	144.68	0.52	0.00
POST pay_order (+400ms `ts-execute-service`)	143	0	116.60	741.13	140.00	151.43	0.48	0.00
POST preserve_ticket	422	0	1847.24	34101.65	7600.00	9011.94	1.41	0.00
POST preserve_ticket (+100ms `ts-execute-service`)	416	0	1861.39	42894.62	7500.00	9127.17	1.39	0.00
POST preserve_ticket (+400ms `ts-execute-service`)	426	0	1857.05	46237.26	7400.00	8691.49	1.42	0.00
GET query_contacts	434	0	20.68	5485.95	28.00	45.54	1.45	0.00
GET query_contacts (+100ms `ts-execute-service`)	430	0	20.74	147.03	28.00	30.38	1.43	0.00
GET query_contacts (+400ms `ts-execute-service`)	435	0	21.87	1853.31	28.00	45.48	1.45	0.00

Table 7: Request statistics for multiple scenarios using 21 users with `ts-execute-service` increased latency

Method	25%	50%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%
GET cancel_order	130	140	150	150	160	220	250	260	360	360	360
GET cancel_order (+100ms `ts-execute-service`)	130	140	150	150	160	190	210	220	250	250	250
GET cancel_order (+400ms `ts-execute-service`)	130	140	150	150	160	170	190	220	320	320	320
GET collect_ticket	70	75	85	89	95	110	160	300	300	300	300
GET collect_ticket (+100ms `ts-execute-service`)	170	180	180	190	190	210	280	310	310	310	310
GET collect_ticket (+400ms `ts-execute-service`)	470	480	480	490	500	530	710	1400	4100	4100	4100
PUT create_consign	4100	4300	4600	4700	4900	5000	5500	5700	8900	8900	8900
PUT create_consign (+100ms `ts-execute-service`)	4100	4400	4700	4800	5000	5300	5500	5500	5500	5500	5500
PUT create_consign (+400ms `ts-execute-service`)	4100	4400	4700	4800	5200	5300	5400	6500	8500	8500	8500
GET enter_station	71	76	86	93	110	200	300	1800	1800	1800	1800
GET enter_station (+100ms `ts-execute-service`)	170	180	180	190	190	200	230	250	310	310	310
GET enter_station (+400ms `ts-execute-service`)	470	480	480	480	490	500	510	550	570	570	570
GET get_assurance_types	23	24	27	28	33	45	56	63	1800	1800	1800
GET get_assurance_types (+100ms `ts-execute-service`)	22	24	27	27	32	37	50	56	68	68	68
GET get_assurance_types (+400ms `ts-execute-service`)	22	24	27	28	33	45	200	570	1300	1300	1300
GET get_food_types	1500	1700	2000	2100	2400	2600	3400	6500	8800	8800	8800
GET get_food_types (+100ms `ts-execute-service`)	1600	1800	2400	2400	2500	2600	3100	3400	3800	3800	3800
GET get_food_types (+400ms `ts-execute-service`)	1500	1700	2000	2100	2400	2600	2700	3000	3700	3700	3700
POST get_order_information	33	35	39	40	51	66	89	160	4000	4000	4000
POST get_order_information (+100ms `ts-execute-service`)	33	35	39	40	44	50	66	86	370	370	370
POST get_order_information (+400ms `ts-execute-service`)	33	35	40	41	51	78	610	1400	3900	3900	3900
POST get_trip_information	300	410	450	470	570	690	1000	1600	8600	8600	8600
POST get_trip_information (+100ms `ts-execute-service`)	290	390	430	440	490	530	600	670	890	890	890
POST get_trip_information (+400ms `ts-execute-service`)	300	400	460	490	570	670	820	970	50000	50000	50000
POST pay_order	130	140	150	150	160	210	260	360	580	580	580
POST pay_order (+100ms `ts-execute-service`)	130	140	150	150	170	190	230	290	300	300	300
POST pay_order (+400ms `ts-execute-service`)	130	140	150	150	160	180	240	590	740	740	740
POST preserve_ticket	4600	7600	12000	13000	16000	20000	24000	28000	34000	34000	34000
POST preserve_ticket (+100ms `ts-execute-service`)	5200	7600	12000	13000	16000	21000	29000	31000	43000	43000	43000
POST preserve_ticket (+400ms `ts-execute-service`)	5100	7400	11000	12000	16000	19000	23000	25000	46000	46000	46000
GET query_contacts	27	28	31	31	34	44	61	76	5500	5500	5500
GET query_contacts (+100ms `ts-execute-service`)	26	28	31	32	36	43	56	98	150	150	150
GET query_contacts (+(+400ms `ts-execute-service`) `ts-execute-service`)	27	28	31	31	36	47	130	410	1900	1900	1900

Table 8: Response time distribution for multiple scenarios using 21 users with `ts-execute-service` increased latency

The tests show that, although the performance of flows that directly invoke the `ts-execute-service` was significantly impacted, especially under the 400ms delay, no other flows experienced degradation. For instance, the GET cancel_order operation maintained a stable median response time of **140 ms** across all scenarios. Similarly, no increase in failures was observed for any request, demonstrating that the delays did not trigger fault conditions in unrelated services. Other flows, such as GET get_assurance_types, preserved their median response time at **24 ms** across all scenarios, and even with a 400 ms delay, its average response time only increased slightly from **30.41 ms** to **36.88 ms**, which is practically insignificant and can be considered environmental oscillation. These examples support the

hypothesis that isolated dependencies have limited influence on the broader environment. Since the `ts-execute-service` is not a transitive dependency for other applications and is only triggered through one route in the system, its degradation does not propagate to the rest of the architecture. This reinforces the idea that the impact of a service remains contained when its dependency degree is low, as few or no services rely on it to function.

4.8 Considerations

During the execution of these experiments, several important aspects regarding dependencies and communication patterns between services emerged. First of all, it is noticeable that the dependency degree calculated by our strategy can change significantly depending on which service experiences latency injection. This behavior simulates real-world situations where specific services become overloaded, resulting in backpressure and altering the usual request flows within the system.

In practical scenarios, such bottlenecks are frequent and strategies are usually implemented to mitigate their effects. Examples include traffic redirection, fallback implementations, and circuit breakers, all of which inherently alter communication patterns among services. Alongside these mitigation strategies, external factors like seasonal variations further contribute to the dynamism of communication patterns. This emphasizes an essential limitation of static dependency mappings – they fail to capture and adapt to these real-time fluctuations.

However, the strategy presented here demonstrates robustness precisely due to its dynamic nature. By continuously recalculating dependency graphs based on recent communication data, it ensures an accurate representation of current service interactions. This dynamic recalculation is vital for securing accurate visibility in highly volatile microservices environments, thus supporting more informed and timely management decisions.

Additionally, our testing validated the correctness of our dependency mapping approach. Specifically, we demonstrated that injecting latency into various services distinctly affected the overall environment, reflecting their calculated dependency degrees. It became evident that services with higher degrees of dependency had a more pronounced impact, especially under significant latency conditions. On the contrary, services with fewer dependencies caused

less systemic disruption when subjected to similar latency levels.

Moreover, our experiments highlighted additional insights. The cascading effects observed when injecting latency into highly interconnected services clearly demonstrated the risk of widespread system degradation due to localized issues. In contrast, systems with well-defined dependency boundaries and lower interdependency exhibited greater resilience, emphasizing the need for careful architectural design and proactive management strategies.

Another significant conclusion from our tests is related to the timing and magnitude of service scaling. Traditional autoscaling mechanisms that rely solely on resource usage metrics (CPU and memory) may inadequately address issues caused by inter-service communication bottlenecks. Incorporating dependency degree metrics, as demonstrated by our experiments, provides a more precise and effective means for timely scaling decisions, potentially avoiding unnecessary resource allocation and mitigating cascading performance impacts.

Maintaining a real-time dependency graph enables the identification, at runtime, of the execution paths that receive the heaviest traffic and the services that occupy the most central positions in those paths – that is, the most critical services of the system. Through these discoveries, it is possible to manage the critical systems in different ways to ensure their performance and availability. Armed with this visibility, custom-targeted resource-management strategies can be developed —- such as pre-allocating additional capacity or increasing the replica count —— to preserve latency objectives and availability targets.

Finally, the experiments were executed with a single observation window t (5 minutes, expressed by the $[5m]$ range vector in the used Prometheus queries) and the graphs built with one or two values for the maximum dependency-graph depth d. Because of this choice, the study does not expose how shorter time windows (capturing sub-minute spikes) or much longer windows (averaging hourly trends) might change the resulting graphs. One systematic exploration of multiple t and d values therefore remains a promising avenue for future work, potentially improving the sensitivity of the mapping to quick bursts while also covering long-term structural patterns.

These findings emphasize the value of dynamic, real-time dependency mapping as a critical factor in efficiently handling microservices-based systems, especially when aiming for adaptability, optimal performance, and resilience.

5 Final considerations

Throughout this work, the proposed approach for mapping microservices dependencies has been developed, analyzed, and validated through experiments and conceptual evaluations. This section consolidates the key insights obtained, discusses observed limitations and possible extensions, and presents an overall reflection on the contributions and future directions. The goal is to summarize the lessons learned, critically assess the system's current capabilities, and point out opportunities for further improvements that can expand the relevance and applicability of the solution.

5.1 Limitations and Additional Considerations

The tests conducted so far demonstrate that the dependency degree approach provides a valid method for understanding how services depend on each other in microservices architectures. However, it is essential to acknowledge certain limitations that can influence both the accuracy and the granularity of these results.

First, accurately interpreting real traffic flows is inherently more complex than merely counting the number of requests or connections. Different endpoints, methods, protocols, and request payloads may behave in ways that cannot be fully captured by aggregated counts alone. Although aggregating metrics at the service level (rather than per-endpoint level) offers a useful approximation – particularly within microservices where each service tends to focus on a limited scope – there is still considerable scope for refinement and deeper granularity.

A good improvement would be to extend the current approach to handle endpoint-level data, potentially by using distributed tracing or alternative strategies (e.g., collecting correlation IDs) to explicitly map which endpoints depend on which others. This would allow the analysis to go beyond "service A calls service B" and delve into "endpoint A1 calls endpoint B3", enriching the overall dependency graph.

Additionally, many real-world systems rely on external third-party services – such as payment gateways, banks, or social-media integrations – which are outside the visibility scope of typical in-cluster monitoring solutions. While these dependencies can be added conceptu-

ally as single abstracted nodes in the graph, the inability to inspect external communication patterns can restrict the fidelity of any downstream analysis, and more robust strategies to deal with this scenario are open to investigation.

Another limitation identified during experimentation is the relatively superficial exploration of dependency graph depths. Currently, the system primarily uses low dependency levels for analysis, which may overlook critical indirect dependencies that could significantly influence system behavior. Future work should address this limitation by systematically exploring deeper dependency graph levels, thus ensuring that scaling decisions incorporate a more holistic understanding of complex microservice interactions.

Lastly, the current research focuses on a single Kubernetes cluster. Production environments often use multiple clusters or even multiple infrastructure environments (e.g., multi-cloud). Future solutions could incorporate cross-cluster networking tools (e.g., Cilium's Cluster Mesh) or ad-hoc aggregations that unify flow data across distinct clusters. For the current proposal, multiple clusters could see each other as black boxes, similar to external dependencies. Although such scenarios are outside the immediate scope of this work and need validation, they represent a promising direction for further investigation.

5.2 Systems Independence

From an architectural standpoint, the solution can be conceptually divided into three layers:

- **Data layer:** responsible for collecting network and system metrics. In the current implementation, this role is fulfilled by Cilium/Hubble, which continuously captures flows at Layers 4 and 7.
- **Processing layer:** consists of data parsing applications like the `metrics-parser-api` and the `dependency-calculator`, which retrieve raw data from the Data Layer and transform it into higher-level metrics such as requests and dependency graphs.
- **Ingestion layer:** Provides storage for metrics and facilitates querying or visualization and consists in Prometheus in the current design.

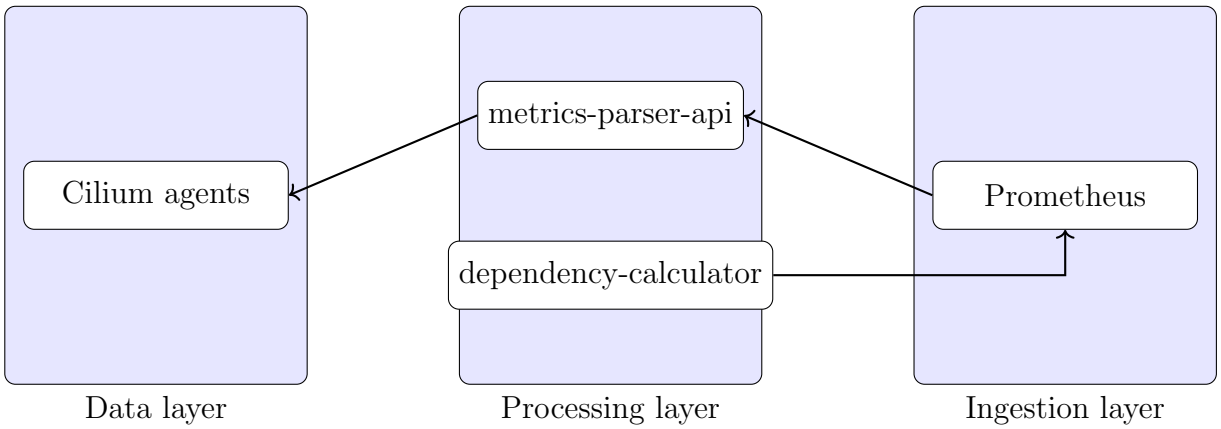


Figure 28: Microservices dependency mapping architecture in a Kubernetes cluster

The objective of the current architecture, which can be visualized in the Figure 28, was to divide each layer into a modular component so that each component can be easily replaced with different technologies. For instance, the data layer could use other Cilium-based flows, such as those from Hubble events subscription or Hubble Flows Exporter. However, it is not limited to Cilium; technologies like Coroot, which can provide the same metrics as Cilium/Hubble, could be used as well. Additionally, service mesh data from platforms such as Istio, Consul, or Linkerd could also be integrated.

The processing layer also can be adapted to handle various types of data. Custom algorithms that consider not only requests but also application-level metrics – such as latency or resource usage as aforementioned – can be implemented within this layer.

Lastly, the ingestion layer can also be replaced as needed. Since Prometheus implements the OpenMetrics protocol, any database that supports this protocol would require no changes in the other layers, including databases like VictoriaMetrics, InfluxDB, and others. Other systems that use alternative storage approaches, such as traditional transactional databases or cloud-based services, can also be integrated by making minor modifications in the processing layer to ensure proper communication.

5.3 Future Extensions

As discussed in the literature, autoscaling decisions in microservices can benefit from additional contextual information. In particular, combining request metrics and dependency

graphs with resource-related metrics (e.g., CPU, memory, or request latency) can enable more effective scaling policies. For instance, detecting backpressure issues or slowdowns in upstream services may allow autoscalers to proactively scale the correct application and avoid over-scaling a downstream service that is not the true bottleneck. This can mitigate cascading overloads and reduce the risk of "wrong scaling" (that is, scaling the wrong application and exacerbating an existing bottleneck) as referred to in earlier sections.

Protocols can also be investigated. Currently, the system focuses on TCP flows (without keepalive) and HTTP/1.1. Further effort is required to extend support to additional protocols such as UDP, gRPC, HTTP/2, HTTP/3, or specialized protocols like TChannel. Native support for additional telemetry endpoints or custom metrics would also enrich the Processing Layer's capabilities, making it more adaptable to varied cloud-native environments and different communication patterns.

In the processing layer, the `metrics-parser-api` arose from a need to work around a known Cilium-related issue. If this issue is resolved upstream, the architecture might become simpler by offloading parsing responsibilities to default Cilium/Hubble flows or by aggregating traffic data with alternative solutions. Additionally, in scenarios involving multiple data sources or more detailed endpoint-level telemetry, the Processing Layer can become a central point for advanced correlation, anomaly detection, or even machine-learning-driven analysis.

Additionally, several other research directions remain open:

- **Security and policy enforcement:** integrating dynamic dependency graphs with real-time policy engines to detect and block anomalous communication patterns.
- **Intra-cluster optimizations:** linking the dependency graph with scheduling and placement algorithms to optimize inter-service latency and reduce cross-node or cross-availability-zones traffic.
- **Hybrid or multi-cluster environments:** extending the solution to manage and visualize dependencies across multiple clusters or multi-cloud setups.
- **Enhanced eBPF usage:** migrating protocol mapping logic from external proxies (e.g., Envoy) into eBPF and storing request counts in eBPF maps, lowering overhead,

improving real-time analysis, and offering deeper visibility into network behavior without relying on any proxies.

- **Dependency graph depth exploration:** investigating the effects of varying depth levels for dependency graphs to gain a more comprehensive understanding of indirect dependencies and their implications for service scaling and overall system resilience.

In essence, while the current architecture demonstrates the viability of collecting and interpreting dependency graph data, numerous research paths remain to enhance its scope, granularity, and adaptability. By systematically exploring these potential extensions, future work can help bridge the gap between simple dynamic flow analysis to more intelligent orchestration and management strategies of microservices at scale.

5.4 Conclusion

The study presented in this work focused on developing an automated approach to mapping service dependencies in Kubernetes-based microservices, providing greater visibility into microservices communication patterns without the need for intrusive instrumentation. By integrating Cilium and its Hubble subsystem, the methodology explored Layer 4 and Layer 7 network flows, building dependency graphs that accurately reflect real-time interactions among services.

A thorough exploration of the academic and industry landscape preceded the work development, covering both pioneering microservices research and more recent publications tackling network observability. This extensive literature review not only demonstrated the ever-growing popularity of microservices but also highlighted the complexities inherent in network-level visibility, distributed tracing, adaptive scaling and general environments management. By comparing multiple established approaches, it became possible to identify clear advantages in gathering real-time data through eBPF-based telemetry, opening new avenues for combining environments management with finer-grained observability techniques.

Translating raw Hubble data into a graph model and subsequently deriving valuable dependency metrics is one of the key contributions of the work. The degree of dependencies theory was proved as a valid approach to quantify and classify both direct and indirect service

relationships. By using this approach, practitioners can better identify bottlenecks that influence entire call chains, such as hidden backpressure effects or wrong scaling decisions. The evolving dependency structures under variable workloads demonstrated that the mapping remains stable, scalable, and sufficiently expressive to guide system enhancements – from adjusting autoscalers to revisiting design choices that propagate unexpected latency spikes.

The proposed mapping strategy, and the resulting request and dependency graphs, were able to consistently provide accurate insights about the actual state of each system across various scenario experiments. By capturing and processing real-time communication flows, these graphs provided an accurate visualization of how different services interact with each other and highlighted where resource bottlenecks were most likely to occur. Hence, the results enhance the importance of synchronizing environment management methods with microservices context in order to enhance system performance and general reliability.

In addition, the experiments showed that injected latency in different levels of the services chain can have a ripple effect throughout the entire ecosystem. Minor delays in a critical service, for instance, showed cascading performance degradation problems across the entire environment, and proved the business need for real-time visibility across microservices dependencies.

The proposed approach is notably lighter in overhead compared to classical instrumentation approaches reliant on sidecars, code changes, or partial sampling, since it takes advantage of network native features for getting the communications data. The Hubble-based collection mechanism provided high-fidelity insights into service call volumes, response times, and error patterns, enabling near real-time dependency mapping. Such near real-time updates facilitate proactive responses to anomalies and quick root cause identification in multiple services environments.

However, the study also highlighted natural limitations. First, all experiments were conducted within Kubernetes clusters where Cilium was available at the infrastructure level, and extensions or portability for other platforms need to be investigated. Second, while the Train Ticket application is large enough to mimic real distributed systems, it is self-contained and lacks the real-world scenarios where integrations between multiple environments – sometimes even with external partners – are needed. Third, the method focuses on network flows and

does not natively incorporate the semantic details of each request (e.g., the functional meaning of Layer 7 calls) or consider additional resources utilization metrics. In future efforts, coupling the presented approach with other data sources and expanding it to multiple environment scenarios can increase the robustness of the proposal and make it more applicable to generic environments

Looking ahead, the modular design of the mapping engine makes it receptive to further extensions. Replacing any of the data, processing or ingestion layer with other technologies can enable a new range of researches: security and policy enforcement, intra-cluster optimizations, multi-cluster environments and others explorations.

The findings of this research highlight the viability and advantages of a non-intrusive, network-level approach to microservices observability in Kubernetes. By reducing development overhead while providing real-time, accurate graphs of service interactions, the proposed method addresses well-known challenges surrounding distributed tracing, manual instrumentation, and mapping strategies. The work finalizes anticipating that automated mapping will become a foundational concept in how modern DevOps teams manage microservices performance and resilience.

References

[1] Akamai. The state of online retail performance — spring 2017, 2017.

[2] Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154, 2018.

[3] Amazon Web Services. Aws: Dynamic scaling for amazon ec2 auto scaling. https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scale-based-on-demand.html.

[4] Chaos Mesh Authors. Chaos mesh: A powerful chaos engineering platform for kubernetes, 2024. Available at https://chaos-mesh.org/.

[5] Cilium Authors. Envoy - cilium documentation, 2024. Available at https://docs.cilium.io/en/latest/security/network/proxy/envoy/.

[6] Coroot Authors. Coroot is an open-source apm observability tool, a datadog and newrelic alternative, 2024. Available at https://coroot.com/.

[7] Deepflow Authors. Deepflow: Instant observability for cloud ai applications, 2025. Available at https://deepflow.io/.

[8] Andrzej Marian Barczak and Michał Barczak. Performance comparison of monolith and microservices based applications. In Nagib et al. Callaos, editor, *Proceedings of the 25th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI 2021), July 18-21, 2021 - Virtual Conference. Volume 1.*, page 120–125. International Institute of Informatics and Systemics, 2021.

[9] Marcus Basalla, Johannes Schneider, Martin Luksik, Roope Jaakonmäki, and Jan vom Brocke. On latency of e-commerce platforms. *Journal of Organizational Computing and Electronic Commerce*, 31:1–17, 01 2021.

[10] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022.

[11] Vincent Bushong, Amr Abdelfattah, Abdullah Maruf, Dipta Das, Austin Lehman, Eric Jaroszewski, Michael Coffey, Tom Černý, Karel Frajták, Pavel Tisnovsky, and Miroslav Bures. On microservice analysis and architecture evolution: A systematic mapping study. *Applied Sciences*, 11:7856, 08 2021.

[12] J. Chen, T. Du, and G. Xiao. A multi-objective optimization for resource allocation of emergent demands in cloud computing. *Journal of Cloud Computing*, 10(1):1–17, 2021.

[13] Shuang Chen, Shay GalOn, Christina Delimitrou, Srilatha Manne, and José F. Martínez. Workload characterization of interactive cloud services on big and small server platforms. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 125–134, 2017.

[14] Cilium Development Team. Hubble: Network, Service Security Observability for Kubernetes. Available at https://github.com/cilium/hubble. Accessed on 07-2023.

[15] PPTAM Contributors. Pptam tool: Production and performance testing based application monitoring, 2024. Available at https://github.com/pptam/pptam-tool.

[16] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery.

[17] D. Hardt, Ed. The OAuth 2.0 Authorization Framework, 2012. Available at https://tools.ietf.org/html/rfc6749.

[18] Saulo S. de Toledo, Antonio Martini, and Dag I.K. Sjøberg. Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study. *Journal of Systems and Software*, 177:110968, 2021.

[19] DeathStarBench. Proposed changes to deathstarbench codebase, 2024. Available at https://github.com/delimitrou/DeathStarBench/pulls?q=is%3Apr+is%3Aclosed+author%3Aalcidesmig.

[20] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.*, 48(4):77–88, mar 2013.

[21] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *SIGPLAN Not.*, 49(4):127–144, feb 2014.

[22] Christina Delimitrou and Christos Kozyrakis. Bolt: I know what you did last summer... in the cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 97–110, New York, NY, USA, 2015. Association for Computing Machinery.

[24] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017.

[25] The Apache Software Foundation. Apache skywalking, 2024. Available at https://skywalking.apache.org/.

[26] Martin Fowler. Microservices: a definition of this new architectural term, 2023. Available at https://martinfowler.com/articles/microservices.html.

[27] FudanSELab. Train ticket: A benchmark microservice system. Available at https://github.com/FudanSELab/train-ticket.

[28] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi,

Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 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*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Alim Ul Gias, Giuliano Casale, and Murray Woodside. Atom: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1994–2004, 2019.

[30] Google. 2022 state of devops report, 2022. Available at https://cloud.google.com/devops/state-of-devops.

[31] Grafana Labs. An implementation of auto-instrumentation using the OpenTelemetry Operator., 2023. Available at https://opentelemetry.io/docs/kubernetes/operator/automatic/.

[32] Magazine Luiza group. Magalu cloud: Platform and cloud services, 2024. Available at https://magalu.cloud/.

[33] Wilhelm Hasselbring and Guido Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246, 2017.

[34] Paul C. Hershey, Shrisha Rao, Charles B. Silio, and Akshay Narayan. System of systems for quality-of-service observation and response in cloud computing environments. *IEEE Systems Journal*, 9(1):212–222, 2015.

[35] Yang Hu, Cees Laat, and Zhiming Zhao. Optimizing service placement for microservice architecture in clouds. *Applied Sciences*, 9:4663, 11 2019.

[36] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, May 2018.

[37] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. Challenges when moving from monolith to microservice architecture. In Irene Garrigós and Manuel Wimmer, editors, *Current Trends in Web Engineering*, pages 32–47, Cham, 2018. Springer International Publishing.

[38] Kubernetes. After learning the ropes with a kubernetes distribution, booking.com built a platform of its own. https://kubernetes.io/case-studies/booking-com/.

[39] Kubernetes. Building an image trust service on kubernetes with notary and tuf. https://kubernetes.io/case-studies/ibm/.

[40] Kubernetes. Kubernetes: Horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[41] Kubernetes. Launching and scaling up experiments, made simple. https://kubernetes.io/case-studies/openai/.

[42] Kubernetes. Nokia: Enabling 5g and devops at a telecom company with kubernetes. https://kubernetes.io/case-studies/nokia/.

[43] Kubernetes. Spotify: An early adopter of containers, spotify is migrating from home-grown orchestration to kubernetes. https://kubernetes.io/case-studies/spotify/.

[44] Kubernetes. Turning to containerization to support millions of rideshares. https://kubernetes.io/case-studies/blablacar/.

[45] Kubernetes. How we architected and run kubernetes on openstack at scale at yahoo! japan, 2016. https://kubernetes.io/blog/2016/10/kubernetes-and-openstack-at-yahoo-japan/.

[46] Grafana Labs. Grafana loki. https://grafana.com/oss/loki/, 2023. Accessed on 08-2023.

[47] Grafana Labs. Grafana, 2024. Available at https://grafana.com/.

[48] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[49] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 450–462, 2015.

[50] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Chengzhong Xu. An in-depth study of microservice call graph and runtime performance. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3901–3914, 2022.

[51] Muhammad Faraz Manzoor, Adnan Abid, Shoaib Farooq, Naeem Ahmed, and Uzma Farooq. Resource allocation techniques in cloud computing: A review and future directions. *Elektronika ir Elektrotechnika*, 26:40–51, 12 2020.

[52] Nabor C. Mendonça, Craig Box, Costin Manolache, and Louis Ryan. The monolith strikes back: Why istio migrated from microservices to a monolithic architecture. *IEEE Software*, 38(5):17–22, 2021.

[53] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 69–84, New York, NY, USA, 2013. Association for Computing Machinery.

[54] Francisco Ponce, Gastón Márquez, and Hernán Astudillo. Migrating from monolithic architecture to microservices: A rapid review. In *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–7, 2019.

[55] Issaret Prachitmutita, Wachirawit Aittinonmongkol, Nasoret Pojjanasuksakul, Montri Supattatham, and Praisan Padungweang. Auto-scaling microservices on iaas under sla with cost-effective framework. In *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*, pages 583–588, 2018.

[56] Prometheus Team. Prometheus Documentation: Overview, Accessed on 06-2023. Available at https://prometheus.io/docs/introduction/overview/.

[57] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Firm: An intelligent fine-grained resource management framework for slo-oriented microservices. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[58] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *CoRR*, abs/1609.09224, 2016.

[59] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should I trust you?": Explaining the predictions of any classifier. *CoRR*, abs/1602.04938, 2016.

[60] Chris Richardson. Microservices io website. https://microservices.io/, 2022. Accessed on 08-2023.

[61] Krzysztof Rzadca, Paweł Findeisen, Jacek Świderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Paweł Krzysztof Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at google scale. In *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[62] Alexis Saransig and Freddy Tapia Leon. *Performance Analysis of Monolithic and Micro Service Architectures – Containers Technology: Proceedings of the 7th International Conference on Software Process Improvement (CIMPS 2018)*, pages 270–279. 01 2019.

[63] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013.

[64] Amazon Web Services. Microservices - aws. https://aws.amazon.com/microservices/, 2022. Accessed on 08-2023.

[65] Anjum Sheikh and Asha Ambhaikar. Quality of services parameters for architectural patterns of iot. *Journal of Information Technology Management*, 13(Special Issue: Big Data Analytics and Management in Internet of Things):36–53, 2021.

[66] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[67] Mehmet Söylemez, Bedir Tekinerdogan, and Ayça Kolukısa Tarhan. Challenges and solution directions of microservice architectures: A systematic literature review. *Applied Sciences*, 12(11), 2022.

[68] Davide Taibi and Valentina Lenarduzzi. On the definition of microservice bad smells. *IEEE Software*, 35(3):56–62, 2018.

[69] Freddy Tapia, Miguel Ángel Mora, Walter Fuertes, Hernán Aules, Edwin Flores, and Theofilos Toulkeridis. From monolithic systems to microservices: A comparative study of performance. *Applied Sciences*, 10(17), 2020.

[70] Elastic Team. Elastic stack. https://www.elastic.co/elastic-stack, 2023. Accessed on 08-2023.

[71] The Cilium Team. Hubble metrics, 2024. Available at https://docs.cilium.io/en/v1.15/observability/metrics/#context-options.

[72] The Istio Authors. Istio, 2023. Available at https://istio.io/.

[73] The Jaeger Authors. Jaeger: open-source, distributed tracing platform, 2023. Available at https://www.jaegertracing.io/.

[74] The OpenTelemetry Authors. Introducing Grafana Beyla: open source ebpf auto-instrumentation for application observability, 2023. Available at https://grafana.com/blog/2023/09/13/grafana-beyla-open-source-ebpf-auto-instrumentation.

[75] The OpenTelemetry Authors. OpenTelemetry, 2023. Available at https://opentelemetry.io/.

[76] Jolly Upadhyaya and Neelu Jyoti Ahuja. Quality of service in cloud computing in higher education: A critical survey and innovative model. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pages 137–140, 2017.

[77] Markos Viggiato, Ricardo Terra, Henrique Rocha, Marco Valente, and Eduardo Figueiredo. Microservices in practice: A survey study. 09 2018.

[78] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590, 2015.

[79] Guangba Yu, Pengfei Chen, and Zibin Zheng. Microscaler: Automatic scaling for microservices with an online learning approach. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 68–75, 2019.

[80] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 167–181, New York, NY, USA, 2021. Association for Computing Machinery.

[81] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling WeChat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, oct 2018.

Appendices

This appendix provides the artifacts necessary to reproduce the study. To minimize space, the code listings have been condensed, prioritising simplicity over exhaustive documentation and modular structure. Fully commented, modular implementations are available in the corresponding GitHub repository.

A Request statistics and response time distributions for tests with multiple scenarios using 12, 21 and 51 users

The tables 9, 10, 11, 12, 13, and 14 demonstrates how the environment how each flow (Method) behaved in load tests with 12, 21 and 51 virtual users.

Method	Requests	Failures	Min (ms)	Max (ms)	Median (ms)	Average (ms)	Current RPS	Failures/s	#Users	Run
GET cancel_order	171	0	110.2946	178.7472	130.0	133.5913	0.5701	0.0	12	1
GET collect_ticket	162	0	51.6351	136.1058	77.0	78.8026	0.5401	0.0	12	1
PUT create_consign	130	130	847.3906	1311.5426	1100.0	1106.0588	0.4334	0.4334	12	1
GET enter_station	162	0	53.8633	138.1663	77.0	79.2011	0.5401	0.0	12	1
GET get_assurance_types	465	0	18.1061	87.3489	25.0	27.3982	1.5502	0.0	12	1
GET get_food_types	462	0	704.9131	3704.7255	1600.0	1752.6336	1.5402	0.0	12	1
POST get_order_information	787	0	25.0180	110.9226	36.0	37.5513	2.6237	0.0	12	1
POST get_trip_information	926	0	197.1905	978.5920	380.0	374.9054	3.0871	0.0	12	1
POST pay_order	162	0	110.7661	190.6124	130.0	136.2413	0.5401	0.0	12	1
POST preserve_ticket	459	0	1720.0460	8648.8105	3900.0	4099.1448	1.5302	0.0	12	1
GET query_contacts	463	0	21.4243	70.0573	29.0	29.8466	1.5436	0.0	12	1
GET cancel_order	176	0	117.7247	210.7074	150.0	148.0000	0.5867	0.0	12	2
GET collect_ticket	171	0	71.2035	133.5607	86.0	89.3114	0.5700	0.0	12	2
PUT create_consign	95	95	3881.8463	4724.2927	4300.0	4268.0007	0.3167	0.3167	12	2
GET enter_station	171	0	68.1338	133.9916	87.0	89.2845	0.5700	0.0	12	2
GET get_assurance_types	446	0	18.5734	97.3542	24.0	26.4682	1.4867	0.0	12	2
GET get_food_types	443	0	721.2301	3253.1378	1600.0	1694.5553	1.4767	0.0	12	2
POST get_order_information	786	0	24.5830	90.3995	34.0	35.9541	2.6200	0.0	12	2
POST get_trip_information	886	0	217.8786	853.0262	390.0	374.8694	2.9533	0.0	12	2
POST pay_order	171	0	109.6845	203.6586	150.0	151.6191	0.5700	0.0	12	2
POST preserve_ticket	439	0	1450.3654	7748.0372	3900.0	3909.5481	1.4633	0.0	12	2
GET query_contacts	443	0	21.5603	79.1722	28.0	29.1714	1.4767	0.0	12	2
GET cancel_order	166	0	109.2262	349.6148	130.0	135.2496	0.5534	0.0	12	3
GET collect_ticket	163	0	50.2304	125.7694	76.0	77.9424	0.5434	0.0	12	3
PUT create_consign	130	130	843.9851	1367.5404	1100.0	1091.5886	0.4334	0.4334	12	3
GET enter_station	164	0	58.8853	129.1909	75.0	76.6418	0.5468	0.0	12	3
GET get_assurance_types	458	0	17.9540	67.2740	25.0	26.3809	1.5269	0.0	12	3
GET get_food_types	457	0	655.5040	3078.8107	1600.0	1671.3596	1.5236	0.0	12	3
POST get_order_information	785	0	23.8536	273.3534	35.0	37.7127	2.6171	0.0	12	3
POST get_trip_information	922	0	210.9560	774.8739	380.0	373.8226	3.0739	0.0	12	3
POST pay_order	163	0	98.3368	339.2406	130.0	134.9037	0.5434	0.0	12	3
POST preserve_ticket	452	0	1706.1734	9853.8536	4200.0	4248.3754	1.5069	0.0	12	3
GET query_contacts	461	0	21.6011	97.0873	28.0	29.3188	1.5369	0.0	12	3

Table 9: Request statistics for multiple scenario using 12 users

Method	Requests	Failures	Min (ms)	Max (ms)	Median (ms)	Average (ms)	Current RPS	Failures/s	#Users	Run
GET cancel_order	154	0	107.2538	186.4524	130.0	135.8620	0.5134	0.0	21	1
GET collect_ticket	148	0	61.3283	126.7651	75.0	77.9352	0.4934	0.0	21	1
PUT create_consign	139	139	893.3579	1376.8384	1100.0	1088.6099	0.4634	0.4634	21	1
GET enter_station	148	0	58.8637	183.8012	76.0	78.9541	0.4934	0.0	21	1
GET get_assurance_types	447	0	16.5631	262.1688	25.0	26.5713	1.4901	0.0	21	1
GET get_food_types	443	0	219.2852	3974.9247	1600.0	1682.5942	1.4768	0.0	21	1
POST get_order_information	738	0	23.8889	354.4882	35.0	37.8095	2.4602	0.0	21	1
POST get_trip_information	889	0	229.1530	1282.7774	390.0	380.7369	2.9636	0.0	21	1
POST pay_order	149	0	116.3151	189.5087	130.0	136.8623	0.4967	0.0	21	1
POST preserve_ticket	432	2	1797.0234	50010.3864	7000.0	9729.9860	1.4401	0.006667	21	1
GET query_contacts	446	0	20.2626	146.0917	28.0	28.7292	1.4868	0.0	21	1
GET cancel_order	147	0	104.9411	181.0625	130.0	134.5785	0.4901	0.0	21	2
GET collect_ticket	145	0	58.5105	104.8133	76.0	76.8379	0.4835	0.0	21	2
PUT create_consign	149	149	897.9349	1360.5836	1100.0	1102.5492	0.4968	0.4968	21	2
GET enter_station	145	0	59.0882	132.7237	77.0	80.7610	0.4835	0.0	21	2
GET get_assurance_types	445	0	18.4567	75.3479	25.0	26.3014	1.4838	0.0	21	2
GET get_food_types	442	0	658.2566	3869.6605	1600.0	1666.3550	1.4738	0.0	21	2
POST get_order_information	734	0	23.4777	81.3638	34.0	35.9443	2.4474	0.0	21	2
POST get_trip_information	885	0	206.6039	761.6135	380.0	372.3982	2.9509	0.0	21	2
POST pay_order	146	0	109.8087	218.7897	130.0	137.6757	0.4868	0.0	21	2
POST preserve_ticket	430	0	1834.6132	44958.1591	7600.0	9577.3952	1.4338	0.0	21	2
GET query_contacts	443	0	19.4413	72.6353	28.0	29.6295	1.4771	0.0	21	2
GET cancel_order	160	0	110.1128	178.7607	130.0	134.3048	0.5334	0.0	21	3
GET collect_ticket	153	0	55.0295	129.6677	75.0	76.5777	0.5101	0.0	21	3
PUT create_consign	131	131	879.1556	1464.4141	1100.0	1106.4421	0.4367	0.4367	21	3
GET enter_station	153	0	56.5167	137.1239	76.0	79.1811	0.5101	0.0	21	3
GET get_assurance_types	452	0	18.8255	66.3328	25.0	26.7502	1.5069	0.0	21	3
GET get_food_types	445	0	700.9262	3855.8000	1600.0	1681.7721	1.4836	0.0	21	3
POST get_order_information	752	0	25.4126	81.3309	34.0	36.2991	2.5071	0.0	21	3
POST get_trip_information	889	0	213.6338	736.3530	380.0	377.3218	2.9638	0.0	21	3
POST pay_order	153	0	113.6302	201.6324	130.0	137.4024	0.5101	0.0	21	3
POST preserve_ticket	436	0	1805.1758	42208.7973	7800.0	9698.6386	1.4536	0.0	21	3
GET query_contacts	446	0	21.8173	60.3529	28.0	28.4551	1.4869	0.0	21	3

Table 10: Request statistics for multiple scenario using 21 users

Method	Requests	Failures	Min (ms)	Max (ms)	Median (ms)	Average (ms)	Current RPS	Failures/s	#Users	Run
GET cancel_order	95	0	108.1018	178.0805	130.0	134.8132	0.3168	0.0	51	1
GET collect_ticket	117	0	60.2627	135.5682	77.0	79.0897	0.3902	0.0	51	1
PUT create_consign	110	110	874.5797	1324.2143	1100.0	1097.6563	0.3668	0.3668	51	1
GET enter_station	117	0	62.3495	150.7006	77.0	80.8036	0.3902	0.0	51	1
GET get_assurance_types	427	0	17.7929	98.6435	24.0	25.7152	1.4239	0.0	51	1
GET get_food_types	422	0	278.5285	3328.0137	1700.0	1739.0825	1.4073	0.0	51	1
POST get_order_information	556	0	24.3389	124.9017	34.0	36.8608	1.8541	0.0	51	1
POST get_trip_information	852	0	229.0653	892.5990	400.0	407.9739	2.8412	0.0	51	1
POST pay_order	117	0	110.4302	256.4282	130.0	139.1069	0.3902	0.0	51	1
POST preserve_ticket	381	90	16.2233	50057.9093	22000.0	25072.7249	1.2705	0.3001	51	1
GET query_contacts	426	0	20.5014	85.1370	28.0	29.3075	1.4206	0.0	51	1
GET cancel_order	101	0	111.2200	177.4058	130.0	134.8689	0.3367	0.0	51	2
GET collect_ticket	103	0	63.9775	130.8074	76.0	79.1278	0.3434	0.0	51	2
PUT create_consign	95	95	206.5989	1264.2488	1100.0	893.8053	0.3167	0.3167	51	2
GET enter_station	103	0	55.7399	118.3924	76.0	78.6329	0.3434	0.0	51	2
GET get_assurance_types	408	0	18.9103	85.4343	25.0	26.3612	1.3603	0.0	51	2
GET get_food_types	405	0	675.3057	3914.0633	1700.0	1743.5866	1.3503	0.0	51	2
POST get_order_information	505	0	25.4474	305.2787	34.0	36.6903	1.6837	0.0	51	2
POST get_trip_information	813	1	27.5359	1020.3794	390.0	404.9128	2.7106	0.003334	51	2
POST pay_order	103	0	116.1196	173.0663	130.0	135.4472	0.3434	0.0	51	2
POST preserve_ticket	363	95	1923.9815	50030.4942	23000.0	26044.6768	1.2103	0.3167	51	2
GET query_contacts	407	0	20.7497	69.9903	28.0	29.2767	1.3570	0.0	51	2
GET cancel_order	94	0	112.8626	187.7810	130.0	135.7655	0.3135	0.0	51	3
GET collect_ticket	113	0	59.8531	124.4829	78.0	80.8865	0.3769	0.0	51	3
PUT create_consign	98	98	891.8419	1324.5073	1100.0	1101.6639	0.3269	0.3269	51	3
GET enter_station	113	0	55.5664	133.8649	77.0	79.6262	0.3769	0.0	51	3
GET get_assurance_types	428	0	16.9947	127.1449	25.0	27.1950	1.4276	0.0	51	3
GET get_food_types	423	0	638.1537	3298.9837	1700.0	1758.0882	1.4110	0.0	51	3
POST get_order_information	531	0	23.2612	103.1637	34.0	37.0418	1.7712	0.0	51	3
POST get_trip_information	849	0	224.1872	1083.2254	400.0	419.4135	2.8319	0.0	51	3
POST pay_order	113	0	103.7368	193.7025	130.0	136.8841	0.3769	0.0	51	3
POST preserve_ticket	381	101	1754.0621	50028.4800	20000.0	24894.8674	1.2709	0.3369	51	3
GET query_contacts	426	0	19.4160	64.4677	28.0	29.6628	1.4210	0.0	51	3

Table 11: Request statistics for multiple scenario using 51 users

Method	#Users	Run	25%	50%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%
GET cancel_order	12	1	130	130	140	140	150	160	170	180	180	180	180
GET collect_ticket	12	1	73	77	82	84	90	100	110	120	140	140	140
PUT create_consign	12	1	1000	1100	1200	1200	1200	1300	1300	1300	1300	1300	1300
GET enter_station	12	1	72	77	83	86	94	99	110	120	140	140	140
GET get_assurance_types	12	1	23	25	28	29	34	45	57	69	87	87	87
GET get_food_types	12	1	1400	1600	2100	2100	2400	2600	2800	3200	3700	3700	3700
POST get_order_information	12	1	33	36	39	40	45	56	66	73	110	110	110
POST get_trip_information	12	1	300	380	430	440	480	520	580	620	980	980	980
POST pay_order	12	1	130	130	140	150	150	160	170	180	190	190	190
POST preserve_ticket	12	1	3200	3900	4900	5000	5800	6300	7400	7800	8600	8600	8600
GET query_contacts	12	1	27	29	31	32	35	39	53	57	70	70	70
GET cancel_order	12	2	140	150	160	160	170	180	180	190	210	210	210
GET collect_ticket	12	2	82	86	95	96	100	110	120	130	130	130	130
PUT create_consign	12	2	4100	4300	4400	4400	4500	4600	4700	4700	4700	4700	4700
GET enter_station	12	2	81	87	94	96	100	110	120	130	130	130	130
GET get_assurance_types	12	2	22	24	27	28	33	43	53	56	97	97	97
GET get_food_types	12	2	1400	1600	1900	2100	2300	2500	2600	2700	3300	3300	3300
POST get_order_information	12	2	32	34	38	39	43	51	61	66	90	90	90
POST get_trip_information	12	2	290	390	430	440	480	520	560	590	850	850	850
POST pay_order	12	2	140	150	160	160	170	180	200	200	200	200	200
POST preserve_ticket	12	2	3400	3900	4100	4500	5000	5300	5600	6300	7700	7700	7700
GET query_contacts	12	2	26	28	30	31	34	42	54	57	79	79	79
GET cancel_order	12	3	130	130	140	140	150	160	170	180	350	350	350
GET collect_ticket	12	3	72	76	81	84	92	98	110	120	130	130	130
PUT create_consign	12	3	1000	1100	1100	1200	1200	1200	1300	1300	1400	1400	1400
GET enter_station	12	3	70	75	79	82	88	100	110	120	130	130	130
GET get_assurance_types	12	3	23	25	27	28	32	40	51	53	67	67	67
GET get_food_types	12	3	1400	1600	2000	2100	2300	2400	2500	2600	3100	3100	3100
POST get_order_information	12	3	33	35	39	40	46	58	67	72	270	270	270
POST get_trip_information	12	3	300	380	430	440	480	530	580	630	770	770	770
POST pay_order	12	3	130	130	140	140	150	160	180	190	340	340	340
POST preserve_ticket	12	3	3600	4200	5000	5100	5700	6300	7400	8200	9900	9900	9900
GET query_contacts	12	3	26	28	30	30	34	41	52	58	97	97	97

Table 12: Response time distribution for multiple scenarios using 12 users

Method	#Users	Run	25%	50%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%
GET cancel_order	21	1	130	130	140	150	160	170	180	180	190	190	190
GET collect_ticket	21	1	71	75	84	85	92	98	110	110	130	130	130
PUT create_consign	21	1	1000	1100	1200	1200	1200	1200	1300	1300	1400	1400	1400
GET enter_station	21	1	71	76	82	84	96	100	110	110	180	180	180
GET get_assurance_types	21	1	23	25	27	28	32	39	47	57	260	260	260
GET get_food_types	21	1	1400	1600	2000	2100	2300	2500	2900	3200	4000	4000	4000
POST get_order_information	21	1	32	35	38	40	45	56	67	75	350	350	350
POST get_trip_information	21	1	300	390	430	440	490	560	600	640	1300	1300	1300
POST pay_order	21	1	130	130	140	140	150	170	190	190	190	190	190
POST preserve_ticket	21	1	4600	7100	13000	14000	20000	25000	32000	36000	50000	50000	50000
GET query_contacts	21	1	26	28	30	30	33	35	45	58	150	150	150
GET cancel_order	21	2	130	130	140	140	150	160	180	180	180	180	180
GET collect_ticket	21	2	72	76	81	82	85	95	98	100	100	100	100
PUT create_consign	21	2	1100	1100	1200	1200	1200	1200	1300	1300	1400	1400	1400
GET enter_station	21	2	72	77	86	88	100	110	120	120	130	130	130
GET get_assurance_types	21	2	23	25	27	27	30	36	54	59	75	75	75
GET get_food_types	21	2	1400	1600	1900	2000	2200	2400	2500	2700	3900	3900	3900
POST get_order_information	21	2	32	34	37	38	44	51	61	68	81	81	81
POST get_trip_information	21	2	300	380	420	440	470	520	580	620	760	760	760
POST pay_order	21	2	140	150	160	160	170	180	200	200	220	220	220
POST preserve_ticket	21	2	4300	7600	12000	14000	18000	26000	32000	34000	45000	45000	45000
GET query_contacts	21	2	26	28	30	31	34	40	58	68	73	73	73
GET cancel_order	21	3	130	130	140	140	150	160	170	170	180	180	180
GET collect_ticket	21	3	70	75	80	82	89	95	110	120	130	130	130
PUT create_consign	21	3	1000	1100	1200	1200	1200	1300	1300	1300	1500	1500	1500
GET enter_station	21	3	71	76	83	86	99	110	120	140	140	140	140
GET get_assurance_types	21	3	23	25	28	29	34	41	51	58	66	66	66
GET get_food_types	21	3	1400	1600	2000	2100	2300	2400	2500	2700	3900	3900	3900
POST get_order_information	21	3	32	34	38	39	44	53	62	69	81	81	81
POST get_trip_information	21	3	300	380	430	440	490	540	600	640	740	740	740
POST pay_order	21	3	130	130	140	150	150	160	180	190	200	200	200
POST preserve_ticket	21	3	4500	7800	12000	14000	20000	24000	29000	34000	42000	42000	42000
GET query_contacts	21	3	26	28	30	30	32	36	40	53	60	60	60

Table 13: Response time distribution for multiple scenarios using 21 users

Method	#Users	Run	25%	50%	75%	80%	90%	95%	98%	99%	99.9%	99.99%	100%
GET cancel_order	51	1	130	130	140	150	150	160	170	180	180	180	180
GET collect_ticket	51	1	72	77	82	85	94	100	110	110	140	140	140
PUT create_consign	51	1	1000	1100	1200	1200	1200	1300	1300	1300	1300	1300	1300
GET enter_station	51	1	72	77	86	90	99	110	110	120	150	150	150
GET get_assurance_types	51	1	22	24	26	27	29	32	52	56	99	99	99
GET get_food_types	51	1	1400	1700	2100	2200	2300	2400	2600	2700	3300	3300	3300
POST get_order_information	51	1	31	34	37	39	49	62	72	80	120	120	120
POST get_trip_information	51	1	320	400	460	490	560	620	690	720	890	890	890
POST pay_order	51	1	130	130	140	150	160	190	200	200	260	260	260
POST preserve_ticket	51	1	7400	22000	48000	50000	50000	50000	50000	50000	50000	50000	50000
GET query_contacts	51	1	26	28	30	30	35	39	50	62	85	85	85
GET cancel_order	51	2	130	130	140	140	150	160	170	170	180	180	180
GET collect_ticket	51	2	72	76	81	84	95	100	110	110	130	130	130
PUT create_consign	51	2	880	1100	1100	1100	1200	1200	1300	1300	1300	1300	1300
GET enter_station	51	2	72	76	82	83	94	110	110	120	120	120	120
GET get_assurance_types	51	2	23	25	27	28	31	34	46	59	85	85	85
GET get_food_types	51	2	1400	1700	2100	2200	2300	2500	2700	2900	3900	3900	3900
POST get_order_information	51	2	31	34	37	39	45	58	67	72	310	310	310
POST get_trip_information	51	2	320	390	450	470	550	630	740	790	1000	1000	1000
POST pay_order	51	2	130	130	140	140	150	160	170	170	170	170	170
POST preserve_ticket	51	2	8400	23000	50000	50000	50000	50000	50000	50000	50000	50000	50000
GET query_contacts	51	2	26	28	31	31	33	38	53	59	70	70	70
GET cancel_order	51	3	130	140	140	140	150	160	170	190	190	190	190
GET collect_ticket	51	3	75	78	85	88	93	110	120	120	120	120	120
PUT create_consign	51	3	1000	1100	1200	1200	1200	1300	1300	1300	1300	1300	1300
GET enter_station	51	3	72	77	83	85	92	110	120	120	130	130	130
GET get_assurance_types	51	3	23	25	28	29	33	39	57	59	130	130	130
GET get_food_types	51	3	1400	1700	2100	2200	2400	2500	2800	3100	3300	3300	3300
POST get_order_information	51	3	32	34	38	39	47	60	70	77	100	100	100
POST get_trip_information	51	3	320	400	460	490	590	690	810	940	1100	1100	1100
POST pay_order	51	3	130	130	140	150	160	170	180	180	190	190	190
POST preserve_ticket	51	3	4500	7800	12000	14000	20000	24000	29000	34000	42000	42000	42000
GET query_contacts	51	3	27	28	31	32	35	39	45	50	60	60	60

Table 14: Response time distribution for multiple scenarios using 51 users

B Client implementation to read network flows from the Hubble Ring Buffer using Go

The following code demonstrates how to use the Hubble Relay gRPC service to stream live network flows from the Ring Buffer.

Listing 5: Golang code to listen to Hubble Ring Buffer

```
package main

import (
        "context"
        "fmt"
        "io"
        "strings"
        "time"

        "github.com/cilium/cilium/api/v1/flow"
        observerpb "github.com/cilium/cilium/api/v1/observer"
        "google.golang.org/grpc"
        "google.golang.org/grpc/codes"
        "google.golang.org/grpc/credentials/insecure"
        "google.golang.org/grpc/status"
)

const (
        HUBBLE_ADDRESS = "10.102.128.121:80" // Replace with Hubble address
)

func main() {
        ctx := context.Background()

        // Establish connection to the Hubble gRPC endpoint
        conn, err := newConn(ctx, HUBBLE_ADDRESS, 1*time.Minute)
        if err != nil {
                panic(fmt.Errorf("failed to connect to Hubble: %w", err))
        }
        defer conn.Close()

        // Create Hubble observer client
        client := observerpb.NewObserverClient(conn)

        // Define flow filter (can be customized)
        filter := &observerpb.GetFlowsRequest{
                Follow: true, // continuously stream events
                Whitelist: []*flow.FlowFilter{
                        {SourceService: []string{}}, // filter can be expanded as needed
                },
        }

        // Start reading flows
        if err := getFlows(ctx, client, filter); err != nil {
                fmt.Printf("Error reading flows: %v\n", err)
        }
}

// newConn establishes a gRPC connection to the provided target with a timeout
func newConn(ctx context.Context, target string, timeout time.Duration) (*grpc.ClientConn, error) {
        // Remove any "tls://" prefix to ensure compatibility with grpc.Dial
        cleanTarget := strings.TrimPrefix(target, "tls://")

        // Use insecure credentials (can be updated to TLS if needed)
        opts := []grpc.DialOption{grpc.WithTransportCredentials(insecure.NewCredentials())}

        // Apply timeout to context
        ctxWithTimeout, cancel := context.WithTimeout(ctx, timeout)
        defer cancel()

```

```
61          // Dial Hubble
62          conn, err := grpc.DialContext(ctxWithTimeout, cleanTarget, opts...)
63          if err != nil {
64                  return nil, fmt.Errorf("unable to dial gRPC target %s: %w", cleanTarget, err)
65          }
66          return conn, nil
67  }
68
69  // getFlows streams flow events from Hubble and prints them
70  func getFlows(ctx context.Context, client observerpb.ObserverClient, req *observerpb.GetFlowsRequest) error
        {
71          stream, err := client.GetFlows(ctx, req)
72          if err != nil {
73                  return fmt.Errorf("failed to get flows: %w", err)
74          }
75
76          for {
77                  resp, err := stream.Recv()
78
79                  switch {
80                  case err == nil:
81                          // Received a valid flow, process it
82                          if err := writeGetFlowsResponse(resp); err != nil {
83                                  return err
84                          }
85                  case err == io.EOF || err == context.Canceled || status.Code(err) == codes.Canceled:
86                          // End of stream or context canceled
87                          fmt.Println("Stream ended or canceled")
88                          return nil
89                  default:
90                          // Unexpected error
91                          return fmt.Errorf("stream error: %w", err)
92                  }
93          }
94  }
95
96  // writeGetFlowsResponse handles the printing of each flow event
97  func writeGetFlowsResponse(res *observerpb.GetFlowsResponse) error {
98          switch event := res.GetResponseTypes().(type) {
99          case *observerpb.GetFlowsResponse_Flow:
100                 flow := event.Flow
101                 fmt.Printf("(%s) %s - %s -> %s (%s)\n",
102                         flow.GetTime().AsTime().Format(time.RFC3339),
103                         flow.GetIP(),
104                         flow.GetSource(),
105                         flow.GetDestination(),
106                         flow.GetVerdict(),
107                 )
108         case *observerpb.GetFlowsResponse_NodeStatus:
109                 // Optional: print status updates
110         case nil:
111                 // No event data
112         default:
113                 fmt.Printf("Unknown response type: %+v\n", event)
114         }
115         return nil
116 }
```

C Deployment file of the `dependency-calculator` API

This appendix introduces the Golang code of the `dependency-calculator` API, encapsulated to be deployed in a Kubernetes cluster.

Listing 6: Deployment files of the dependency-calculator API

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: dependency-calculator-config
data:
  main.go: |
    package main

    import (
      "context"
      "fmt"
      "log"
      "math"
      "net/http"
      "os/exec"
      "strconv"
      "strings"
      "time"

      "github.com/gin-gonic/gin"
      "github.com/prometheus/client_golang/api"
      v1 "github.com/prometheus/client_golang/api/prometheus/v1"
      "github.com/prometheus/common/model"
      "gonum.org/v1/gonum/graph"
      "gonum.org/v1/gonum/graph/encoding"
      "gonum.org/v1/gonum/graph/encoding/dot"
      "gonum.org/v1/gonum/graph/simple"
    )

    // ------------------------------------------------------------------------
    // Internal graph helpers
    // ------------------------------------------------------------------------

    const (
      intervalToConsider = 5
      prometheusURL      = "http://localhost:9090" // Update to prometheus URL
    )

    // labelled node/edge wrappers so DOT keeps service names instead of int IDs

    type labeledNode struct {
      graph.Node
      label string
    }

    func (n labeledNode) DOTID() string { return n.label }
    func (n labeledNode) Attributes() []encoding.Attribute {
      return []encoding.Attribute{{Key: "label", Value: n.label}}
    }

    type labeledEdge struct {
      simple.WeightedEdge
```

```
53        label string
54    }
55
56    func (e labeledEdge) Attributes() []encoding.Attribute {
57      return []encoding.Attribute{{Key: "label", Value: fmt.Sprintf("%.2f", e.Weight())}}
58    }
59
60    // createGraph fetches dependency data from Prometheus and constructs a directed weighted graph.
61    func createGraph(metric, additionalQueryFilters string, useRatio bool, prometheusURL string,
          excludedServices []string) (*simple.WeightedDirectedGraph, map[string]labeledNode, map[string]
          float64, error) {
62      client, err := api.NewClient(api.Config{Address: prometheusURL})
63      if err != nil {
64        return nil, nil, nil, fmt.Errorf("creating Prometheus client: %w", err)
65      }
66
67      v1api := v1.NewAPI(client)
68      ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
69      defer cancel()
70
71      // 1) total requests received per destination
72      totalQuery := fmt.Sprintf("sum by (destination_workload) (increase(%s%s[%dm]))", metric,
            additionalQueryFilters, intervalToConsider)
73      totalResult, warnings, err := v1api.Query(ctx, totalQuery, time.Now())
74      if err != nil {
75        return nil, nil, nil, fmt.Errorf("querying Prometheus totals: %w", err)
76      }
77      if len(warnings) > 0 {
78        log.Printf("Prometheus warnings: %v", warnings)
79      }
80
81      totalVector, ok := totalResult.(model.Vector)
82      if !ok {
83        return nil, nil, nil, fmt.Errorf("unexpected total result type: %T", totalResult)
84      }
85
86      // 2) requests from A -> B
87      requestsQuery := fmt.Sprintf("sum by (source_workload, destination_workload) (increase(%s%s[%dm]))",
            metric, additionalQueryFilters, intervalToConsider)
88      requestsResult, warnings, err := v1api.Query(ctx, requestsQuery, time.Now())
89      if err != nil {
90        return nil, nil, nil, fmt.Errorf("querying Prometheus requests: %w", err)
91      }
92      if len(warnings) > 0 {
93        log.Printf("Prometheus warnings: %v", warnings)
94      }
95      requestsVector, ok := requestsResult.(model.Vector)
96      if !ok {
97        return nil, nil, nil, fmt.Errorf("unexpected requests result type: %T", requestsResult)
98      }
99
100     // ------------------------------------------------------------------
101     // Build the graph structure
102     // ------------------------------------------------------------------
103     receivedRequests := make(map[string]float64) // destination -> total reqs
104     for _, sample := range totalVector {
105       dest := string(sample.Metric["destination_workload"])
106       receivedRequests[dest] = float64(sample.Value)
107     }
108
109     g := simple.NewWeightedDirectedGraph(0, 0)
110     nodeMap := make(map[string]labeledNode)
```

```
111
112       // unique list of service names (src or dest)
113       servicesList := getUniqueSrcDest(requestsVector)
114       exclusion := make(map[string]bool)
115       for _, s := range excludedServices {
116         exclusion[s] = true
117       }
118
119       servicesList = append(servicesList, "external")
120       for _, svc := range servicesList {
121         if exclusion[svc] || svc == "" {
122           continue
123         }
124         node := labeledNode{Node: g.NewNode(), label: svc}
125         g.AddNode(node)
126         nodeMap[svc] = node
127       }
128
129       // edges
130       for _, sample := range requestsVector {
131         src := string(sample.Metric["source_workload"])
132         dest := string(sample.Metric["destination_workload"])
133
134         if src == "" && dest == "ts-gateway-service" {
135           src = "external"
136         }
137         if src == dest || src == "" || dest == "" || exclusion[src] || exclusion[dest] {
138           continue
139         }
140         if strings.HasSuffix(dest, "mysql") { // skip DB targets
141           continue
142         }
143
144         value := float64(sample.Value)
145         if useRatio {
146           total := receivedRequests[src]
147           if total == 0 {
148             total = 1
149           }
150           value = value / total
151         }
152
153         edge := labeledEdge{
154           WeightedEdge: g.NewWeightedEdge(nodeMap[src], nodeMap[dest], value).(simple.WeightedEdge),
155           label:        fmt.Sprintf("%s -> %s", src, dest),
156         }
157         g.SetWeightedEdge(edge)
158       }
159
160       return g, nodeMap, receivedRequests, nil
161     }
162
163     // calculateDependencyDegreeGraph iterates through all service pairs and calculates dependency scores.
164     func calculateDependencyDegreeGraph(g *simple.WeightedDirectedGraph, receivedRequests map[string]
          float64, depth int, nodeMap map[string]labeledNode) map[string]map[string]float64 {
165       depth--
166       dep := make(map[string]map[string]float64)
167       for src := range receivedRequests {
168         dep[src] = make(map[string]float64)
169         for dst := range receivedRequests {
170           if src == dst {
171             dep[src][dst] = 1
```

```
            continue
          }
          v := computeDependency(g, receivedRequests, src, dst, depth, nodeMap)
          if v != 0 && !math.IsNaN(v) && !math.IsInf(v, 1) {
            dep[src][dst] = v
          }
        }
      }
      return dep
    }

    // computeDependency recursively calculates dependency between services.
    func computeDependency(g *simple.WeightedDirectedGraph, received map[string]float64, src, dst string,
        depth int, nodeMap map[string]labeledNode) float64 {
      srcN, ok1 := nodeMap[src]
      dstN, ok2 := nodeMap[dst]
      if !ok1 || !ok2 {
        return 0
      }

      if e := g.WeightedEdge(srcN.Node.ID(), dstN.Node.ID()); e != nil {
        return e.Weight() / received[src]
      }
      if depth <= 0 {
        return 0
      }

      total := 0.0
      for nbrs := g.From(srcN.Node.ID()); nbrs.Next(); {
        inter := nbrs.Node().(labeledNode)
        if inter.label == dst {
          continue
        }
        total += computeDependency(g, received, src, inter.label, depth-1, nodeMap) *
          computeDependency(g, received, inter.label, dst, depth-1, nodeMap)
      }
      return total
    }

    // ------------------------------------------------------------------------
    // REST API layer using Gin
    // ------------------------------------------------------------------------

    type Edge struct {
      Source string  `json:"source"`
      Target string  `json:"target"`
      Weight float64 `json:"weight"`
    }

    type Node struct {
      ID string `json:"id"`
    }

    type GraphResponse struct {
      Nodes []Node `json:"nodes"`
      Edges []Edge `json:"edges"`
    }

    type ErrorResponse struct {
      Error string `json:"error"`
    }

```

```
233     func main() {
234       r := gin.Default()
235
236       r.GET("/health", func(c *gin.Context) { c.String(http.StatusOK, "ok") })
237       r.GET("/graph", graphHandler)
238       r.GET("/dependency", dependencyHandler)
239       r.GET("/graph/dot", dotHandler)
240       // Optional: PNG endpoint (requires Graphviz installed in the container/host)
241       r.GET("/graph/png", pngHandler)
242
243       log.Println("Dependency API listening on :8080")
244       if err := r.Run(":8080"); err != nil {
245         log.Fatalf("failed to start server: %v", err)
246       }
247     }
248
249     // graphHandler returns nodes & edges for immediate graph visualization (e.g. D3.js)
250     func graphHandler(c *gin.Context) {
251       metric := c.DefaultQuery("metric", "hubble_parsed_http_requests_total")
252       filters := c.DefaultQuery("filters", "{traffic_direction=\"ingress\"}")
253       ratio := parseBool(c.DefaultQuery("ratio", "false"))
254       promURL := c.DefaultQuery("prom", prometheusURL)
255       exclude := parseCSV(c.DefaultQuery("exclude", "nacos,skywalking"))
256
257       g, nodeMap, _, err := createGraph(metric, filters, ratio, promURL, exclude)
258       if err != nil {
259         c.JSON(http.StatusInternalServerError, ErrorResponse{err.Error()})
260         return
261       }
262
263       id2label := make(map[int64]string)
264       nodes := make([]Node, 0, len(nodeMap))
265       for _, n := range nodeMap {
266         nodes = append(nodes, Node{ID: n.label})
267         id2label[n.ID()] = n.label
268       }
269
270       edgesIter := g.Edges()
271       var edges []Edge
272       for edgesIter.Next() {
273         ge := edgesIter.Edge()
274         we, ok := ge.(graph.WeightedEdge)
275         if !ok {
276           continue
277         }
278         edges = append(edges, Edge{
279           Source: id2label[we.From().ID()],
280           Target: id2label[we.To().ID()],
281           Weight: we.Weight(),
282         })
283       }
284
285       c.JSON(http.StatusOK, GraphResponse{Nodes: nodes, Edges: edges})
286     }
287
288     // dependencyHandler returns the dependency matrix in JSON.
289     func dependencyHandler(c *gin.Context) {
290       metric := c.DefaultQuery("metric", "hubble_parsed_http_requests_total")
291       filters := c.DefaultQuery("filters", "{traffic_direction=\"ingress\"}")
292       ratio := parseBool(c.DefaultQuery("ratio", "false"))
293       promURL := c.DefaultQuery("prom", prometheusURL)
294       exclude := parseCSV(c.DefaultQuery("exclude", "nacos,skywalking"))
```

```
295         depthStr := c.DefaultQuery("depth", "1")
296         depth, _ := strconv.Atoi(depthStr)
297         if depth < 1 {
298           depth = 1
299         }
300
301         g, nodeMap, received, err := createGraph(metric, filters, ratio, promURL, exclude)
302         if err != nil {
303           c.JSON(http.StatusInternalServerError, ErrorResponse{err.Error()})
304           return
305         }
306
307         dep := calculateDependencyDegreeGraph(g, received, depth, nodeMap)
308         c.JSON(http.StatusOK, dep)
309       }
310
311       // dotHandler serialises the current graph in Graphviz DOT format.
312       func dotHandler(c *gin.Context) {
313         metric := c.DefaultQuery("metric", "hubble_parsed_http_requests_total")
314         filters := c.DefaultQuery("filters", "{traffic_direction=\"ingress\"}")
315         ratio := parseBool(c.DefaultQuery("ratio", "false"))
316         promURL := c.DefaultQuery("prom", prometheusURL)
317         exclude := parseCSV(c.DefaultQuery("exclude", "nacos,skywalking"))
318
319         g, _, _, err := createGraph(metric, filters, ratio, promURL, exclude)
320         if err != nil {
321           c.JSON(http.StatusInternalServerError, ErrorResponse{err.Error()})
322           return
323         }
324
325         dotData, err := dot.Marshal(g, "WorkloadGraph", "", "  ")
326         if err != nil {
327           c.JSON(http.StatusInternalServerError, ErrorResponse{fmt.Sprintf("marshalling DOT: %v", err)})
328           return
329         }
330
331         c.Data(http.StatusOK, "text/vnd.graphviz", dotData)
332       }
333
334       // pngHandler streams a PNG image generated via Graphviz. Requires `dot` binary.
335       func pngHandler(c *gin.Context) {
336         metric := c.DefaultQuery("metric", "hubble_parsed_http_requests_total")
337         filters := c.DefaultQuery("filters", "{traffic_direction=\"ingress\"}")
338         ratio := parseBool(c.DefaultQuery("ratio", "false"))
339         promURL := c.DefaultQuery("prom", prometheusURL)
340         exclude := parseCSV(c.DefaultQuery("exclude", "nacos,skywalking"))
341
342         g, _, _, err := createGraph(metric, filters, ratio, promURL, exclude)
343         if err != nil {
344           c.JSON(http.StatusInternalServerError, ErrorResponse{err.Error()})
345           return
346         }
347
348         dotData, err := dot.Marshal(g, "WorkloadGraph", "", "  ")
349         if err != nil {
350           c.JSON(http.StatusInternalServerError, ErrorResponse{fmt.Sprintf("marshalling DOT: %v", err)})
351           return
352         }
353
354         cmd := exec.Command("dot", "-Tpng")
355         cmd.Stdin = strings.NewReader(string(dotData))
356         pngBytes, err := cmd.Output()
```

```
      if err != nil {
        c.JSON(http.StatusInternalServerError, ErrorResponse{fmt.Sprintf("running Graphviz: %v", err)})
        return
      }

      c.Data(http.StatusOK, "image/png", pngBytes)
    }

    // ------------------------------------------------------------------------
    // Utility helpers
    // ------------------------------------------------------------------------

    func parseBool(str string) bool {
      b, _ := strconv.ParseBool(str)
      return b
    }

    func parseCSV(in string) []string {
      if in == "" {
        return nil
      }
      parts := strings.Split(in, ",")
      out := make([]string, 0, len(parts))
      for _, p := range parts {
        p = strings.TrimSpace(p)
        if p != "" {
          out = append(out, p)
        }
      }
      return out
    }

    // getUniqueSrcDest builds a list of unique service names appearing as source or destination in the
        vector.
    func getUniqueSrcDest(v model.Vector) []string {
      set := make(map[string]struct{})
      for _, s := range v {
        src := string(s.Metric["source_workload"])
        dst := string(s.Metric["destination_workload"])
        if src != "" {
          set[src] = struct{}{}
        }
        if dst != "" {
          set[dst] = struct{}{}
        }
      }
      out := make([]string, 0, len(set))
      for k := range set {
        out = append(out, k)
      }
      return out
    }

---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: dependency-calculator-deployment
spec:
  replicas: 1
  selector:
```

```
    matchLabels:
      app: dependency-calculator
  template:
    metadata:
      labels:
        app: dependency-calculator
    spec:
      serviceAccountName: dependency-calculator-service-account
      containers:
      - name: dependency-calculator
        image: golang:1.23.0
        command: ["/bin/sh", "-c"]
        args: [
          "mkdir -p /go/src/app && cp /app/main.go /app/lib.go /go/src/app/ && cd /go/src/app && go mod
               init app && go mod tidy && go run ."
        ]
        ports:
        - containerPort: 8080
        volumeMounts:
        - name: go-code
          mountPath: /app
      volumes:
      - name: go-code
        configMap:
          name: dependency-calculator-config

---

apiVersion: v1
kind: Service
metadata:
  name: dependency-calculator-service
spec:
  selector:
    app: dependency-calculator
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

D Deployment file of the `metrics-parser-api` API

This appendix introduces the Golang code of the $metrics-parser-api$ API, encapsulated to be deployed in a Kubernetes cluster.

Listing 7: Deployment files of the metrics-parser-api API

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: go-metrics-parser-config
data:
  main.go: |
    package main

    import (
```

```
10        "context"
11        "fmt"
12        "io/ioutil"
13        "log"
14        "net/http"
15        "strings"
16
17        "github.com/prometheus/client_golang/prometheus"
18        "github.com/prometheus/client_golang/prometheus/promhttp"
19        dto "github.com/prometheus/client_model/go"
20        "github.com/prometheus/common/expfmt"
21        v1 "k8s.io/api/core/v1"
22        metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
23        "k8s.io/client-go/kubernetes"
24        "k8s.io/client-go/rest"
25    )
26
27    type Metric struct {
28        Name   string            `json:"name"`
29        Labels map[string]string `json:"labels"`
30        Value  float64           `json:"value"`
31    }
32
33    var (
34        allPossibleLabels = []string{"destination_ip", "destination_namespace", "destination_workload", "
              method", "protocol", "reporter", "source_ip", "source_namespace", "source_workload", "status", "
              traffic_direction", "flag"}
35
36        hubbleHTTPRequestsTotal = prometheus.NewCounterVec(
37            prometheus.CounterOpts{
38                Name: "hubble_parsed_http_requests_total",
39                Help: "Total number of HTTP requests observed by Hubble",
40            },
41            allPossibleLabels,
42        )
43        hubbleFlowsProcessedTotal = prometheus.NewCounterVec(
44            prometheus.CounterOpts{
45                Name: "hubble_parsed_flows_processed_total",
46                Help: "Total number of flows processed by Hubble",
47            },
48            allPossibleLabels,
49        )
50        hubbleTCPFlagsTotal = prometheus.NewCounterVec(
51            prometheus.CounterOpts{
52                Name: "hubble_parsed_tcp_flags_total",
53                Help: "TCP Flags",
54            },
55            allPossibleLabels,
56        )
57        hubbleHTTPRequestDurationSecondsSum = prometheus.NewCounterVec(
58            prometheus.CounterOpts{
59                Name: "hubble_parsed_http_request_duration_seconds_sum",
60                Help: "Sum of latencies for HTTP requests",
61            },
62            allPossibleLabels,
63        )
64    )
65
66    func init() {
67        prometheus.MustRegister(hubbleHTTPRequestsTotal)
68        prometheus.MustRegister(hubbleFlowsProcessedTotal)
69        prometheus.MustRegister(hubbleHTTPRequestDurationSecondsSum)
```

```
70      prometheus.MustRegister(hubbleTCPFlagsTotal)
71    }
72
73    func main() {
74      http.Handle("/metrics", promhttp.InstrumentMetricHandler(prometheus.DefaultRegisterer, http.
            HandlerFunc(metricsHandler)))
75      log.Fatal(http.ListenAndServe(":8080", nil))
76    }
77
78    func metricsHandler(w http.ResponseWriter, r *http.Request) {
79      // Create Kubernetes client using in-cluster config
80      config, err := rest.InClusterConfig()
81      if err != nil {
82        http.Error(w, fmt.Sprintf("Error loading in-cluster config: %v", err), http.
              StatusInternalServerError)
83        return
84      }
85
86      clientset, err := kubernetes.NewForConfig(config)
87      if err != nil {
88        http.Error(w, fmt.Sprintf("Error creating clientset: %v", err), http.StatusInternalServerError)
89        return
90      }
91
92      // Fetch Cilium daemonset pods
93      pods, err := clientset.CoreV1().Pods("kube-system").List(context.TODO(), metav1.ListOptions{
94        LabelSelector: "k8s-app=cilium",
95      })
96      if err != nil {
97        http.Error(w, fmt.Sprintf("Error listing Cilium pods: %v", err), http.StatusInternalServerError)
98        return
99      }
100
101     plf, err := NewPodLabelFetcher()
102     if err != nil {
103       http.Error(w, fmt.Sprintf("Error creating PodLabelFetcher: %v", err), http.
              StatusInternalServerError)
104       return
105     }
106
107     // Reset the metrics
108     hubbleHTTPRequestsTotal.Reset()
109     hubbleFlowsProcessedTotal.Reset()
110     hubbleHTTPRequestDurationSecondsSum.Reset()
111     hubbleTCPFlagsTotal.Reset()
112
113     // Iterate over pods and fetch metrics
114     for _, pod := range pods.Items {
115       podMetrics, err := fetchMetricsFromPod(pod)
116       if err != nil {
117         fmt.Printf("Error fetching metrics from pod %s: %v", pod.Name, err)
118         continue
119       }
120
121       for _, metric := range podMetrics {
122         // Resolve IPs to service names
123         if metric.Labels["destination_workload"] == "" {
124           serviceName, err := plf.GetPodLabel(metric.Labels["destination_ip"])
125           if err == nil {
126             metric.Labels["destination_workload"] = serviceName
127           }
128         }
```

```
129
130             if metric.Labels["source_workload"] == "" {
131               serviceName, err := plf.GetPodLabel(metric.Labels["source_ip"])
132               if err == nil {
133                 metric.Labels["source_workload"] = serviceName
134               }
135             }
136
137             // Preserve all labels and add default empty values if necessary
138             labels := prometheus.Labels{}
139             for _, label := range allPossibleLabels {
140               labels[label] = metric.Labels[label]
141             }
142
143             // Update Prometheus metrics
144             switch metric.Name {
145             case "hubble_http_requests_total":
146               hubbleHTTPRequestsTotal.With(labels).Add(metric.Value)
147             case "hubble_flows_processed_total":
148               hubbleFlowsProcessedTotal.With(labels).Add(metric.Value)
149             case "hubble_http_request_duration_seconds_sum":
150               hubbleHTTPRequestDurationSecondsSum.With(labels).Add(metric.Value)
151             case "hubble_tcp_flags_total":
152               hubbleTCPFlagsTotal.With(labels).Add(metric.Value)
153             }
154           }
155         }
156
157         promhttp.Handler().ServeHTTP(w, r)
158       }
159
160       func fetchMetricsFromPod(pod v1.Pod) ([]Metric, error) {
161         url := fmt.Sprintf("http://%s:9965/metrics", pod.Status.PodIP)
162         resp, err := http.Get(url)
163         if err != nil {
164           return nil, fmt.Errorf("error fetching metrics from pod %s: %v", pod.Name, err)
165         }
166         defer resp.Body.Close()
167
168         if resp.StatusCode != http.StatusOK {
169           return nil, fmt.Errorf("unexpected status code %d fetching metrics from pod %s", resp.StatusCode,
                pod.Name)
170         }
171
172         body, err := ioutil.ReadAll(resp.Body)
173         if err != nil {
174           return nil, fmt.Errorf("error reading response body from pod %s: %v", pod.Name, err)
175         }
176
177         parser := expfmt.TextParser{}
178         metricFamilies, err := parser.TextToMetricFamilies(strings.NewReader(string(body)))
179         if err != nil {
180           return nil, fmt.Errorf("error parsing metrics from pod %s: %v", pod.Name, err)
181         }
182
183         fmt.Printf("Metrics fetched from pod %s\n", pod.Name)
184
185         // Filter for the required metrics
186         requiredMetrics := []string{
187           "hubble_http_requests_total",
188           "hubble_flows_processed_total",
189           "hubble_http_request_duration_seconds_sum",
```

```
        "hubble_tcp_flags_total",
      }

      metrics := parseMetrics(metricFamilies, requiredMetrics)
      return metrics, nil
    }

    func parseMetrics(metricFamilies map[string]*dto.MetricFamily, requiredMetrics []string) []Metric {
      var metrics []Metric
      requiredMetricSet := make(map[string]struct{})
      for _, metricName := range requiredMetrics {
        requiredMetricSet[metricName] = struct{}{}
      }

      for name, family := range metricFamilies {
        if _, found := requiredMetricSet[name]; !found {
          continue
        }
        for _, m := range family.Metric {
          labels := make(map[string]string)
          for _, label := range m.Label {
            labels[label.GetName()] = label.GetValue()
          }
          var value float64
          if m.Gauge != nil {
            value = m.Gauge.GetValue()
          } else if m.Counter != nil {
            value = m.Counter.GetValue()
          } else if m.Summary != nil {
            value = m.Summary.GetSampleSum() // example for summaries
          } else if m.Histogram != nil {
            value = m.Histogram.GetSampleSum() // example for histograms
          }
          metric := Metric{
            Name:   name,
            Labels: labels,
            Value:  value,
          }
          metrics = append(metrics, metric)
        }
      }

      return metrics
    }

    func resolveIPToService(clientset *kubernetes.Clientset, ip string) (string, error) {
      services, err := clientset.CoreV1().Services("").List(context.TODO(), metav1.ListOptions{})
      if (err != nil) {
        return "", err
      }

      for _, svc := range services.Items {
        if svc.Spec.ClusterIP == ip {
          return svc.Name, nil
        }
      }
      return "", fmt.Errorf("service not found for IP: %s", ip)
    }

  lib.go: |
    package main

```

```
252     import (
253       "context"
254       "fmt"
255       "sync"
256
257       v1 "k8s.io/api/core/v1"
258       metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
259       "k8s.io/client-go/kubernetes"
260
261       "k8s.io/client-go/rest"
262     )
263
264     type PodLabelFetcher struct {
265       clientset *kubernetes.Clientset
266       cache     map[string]string
267       mu        sync.Mutex
268       pods      *v1.PodList
269     }
270
271     func NewPodLabelFetcher() (*PodLabelFetcher, error) {
272       // Create Kubernetes client using in-cluster config
273       config, err := rest.InClusterConfig()
274       if err != nil {
275         return nil, err
276       }
277
278       // Create Kubernetes clientset
279       clientset, err := kubernetes.NewForConfig(config)
280       if err != nil {
281         panic(err.Error())
282       }
283
284       pods, err := clientset.CoreV1().Pods("").List(context.TODO(), metav1.ListOptions{})
285       if err != nil {
286         panic(err)
287       }
288       return &PodLabelFetcher{
289         clientset: clientset,
290         cache:     make(map[string]string),
291         pods:      pods,
292       }, nil
293     }
294
295     func (f *PodLabelFetcher) GetPodLabel(ip string) (string, error) {
296       f.mu.Lock()
297       if label, found := f.cache[ip]; found {
298         f.mu.Unlock()
299         return label, nil
300       }
301       f.mu.Unlock()
302
303       for _, pod := range f.pods.Items {
304         if pod.Status.PodIP == ip {
305           label := pod.Labels["app"]
306           f.mu.Lock()
307           f.cache[ip] = label
308           f.mu.Unlock()
309           return label, nil
310         }
311       }
312
313       return "", fmt.Errorf("pod with IP %s not found", ip)
```

```
    }

    func (f *PodLabelFetcher) GetPodLabels(ips []string) (map[string]string, error) {
      labels := make(map[string]string)
      for _, ip := range ips {
        label, err := f.GetPodLabel(ip)
        if err != nil {
          return nil, err
        }
        labels[ip] = label
      }
      return labels, nil
    }


---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: go-metrics-parser-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: go-metrics-parser
  template:
    metadata:
      labels:
        app: go-metrics-parser
      annotations:
        prometheus.io/port: "8080"
        prometheus.io/scrape: "true"
    spec:
      serviceAccountName: go-metrics-parser-service-account
      containers:
      - name: go-metrics-parser
        image: golang:1.23.0
        command: ["/bin/sh", "-c"]
        args: [
          "mkdir -p /go/src/app && cp /app/main.go /app/lib.go /go/src/app/ && cd /go/src/app && go mod
               init app && go mod tidy && go run ."
        ]
        ports:
        - containerPort: 8080
        volumeMounts:
        - name: go-code
          mountPath: /app
      volumes:
      - name: go-code
        configMap:
          name: go-metrics-parser-config

---

apiVersion: v1
kind: Service
metadata:
  name: go-metrics-parser-service
spec:
  selector:
    app: go-metrics-parser
```

```
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080


---

apiVersion: v1
kind: ServiceAccount
metadata:
  name: go-metrics-parser-service-account
  namespace: default

---

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: go-metrics-parser-cluster-role
rules:
- apiGroups: [""]
  resources: ["pods", "services"]
  verbs: ["get", "list"]

---

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: go-metrics-parser-cluster-role-binding
subjects:
- kind: ServiceAccount
  name: go-metrics-parser-service-account
  namespace: default
roleRef:
  kind: ClusterRole
  name: go-metrics-parser-cluster-role
  apiGroup: rbac.authorization.k8s.io
```

E Kubernetes cluster and applications setup

This appendix reproduces every command used to create the cluster utilized in the experiments. Each subsection corresponds to one bootstrap phase.

E.1 Cluster networking: installing Cilium

Listing 8: Install Cilium 1.15.3 with Hubble, Prometheus, Envoy, and Ingress

```
# The following kubectl patches are needed in case the cluster already contains one non-Helm Cilium
    installation.

# Enumerate resource types to patch
```

```
resource_types=("configmaps" "secrets" "serviceaccounts" "daemonsets" "deployments" "replicasets" "pods" "
    services" "endpoints" "jobs" "cronjobs" "persistentvolumeclaims" "persistentvolumes" "statefulsets" "
    ingresses" "networkpolicies" "roles" "rolebindings" "clusterroles" "clusterrolebindings")
# Loop through each resource type
for resource_type in "${resource_types[@]}"; do
  # Get all resources of this type that start with "cilium" in the kube-system namespace
  resources=$(kubectl get $resource_type -n kube-system -o name | grep -E ".*(cilium|hubble).*")
  # Loop through each resource and apply the patches
  for resource in $resources; do
    kubectl patch $resource -n kube-system --patch '{
      "metadata": {
        "labels": {
          "app.kubernetes.io/managed-by": "Helm"
        }
      }
    }'
    kubectl patch $resource -n kube-system --patch '{
      "metadata": {
        "annotations": {
          "meta.helm.sh/release-name": "cilium"
        }
      }
    }'
    kubectl patch $resource -n kube-system --patch '{
      "metadata": {
        "annotations": {
          "meta.helm.sh/release-namespace": "kube-system"
        }
      }
    }'
  done
done

# Install (or reinstall) cilium using Helm with the required configurations
helm upgrade --install cilium cilium/cilium --version 1.15.3 \
  --namespace kube-system \
  --set prometheus.enabled="true" \
  --set operator.prometheus.enabled="true" \
  --set hubble.enabled="true" \
  --set hubble.ui.enabled="true" \
  --set hubble.metrics.enableOpenMetrics="true" \
  --set hubble.relay.enabled="true" \
  --set hubble.metrics.enabled="{httpV2:exemplars=true;labelsContext=source_ip\,source_namespace\,
      source_workload\,destination_ip\,destination_namespace\,destination_workload\,traffic_direction,dns,
      drop,flow:labelsContext=source_ip\,source_namespace\,source_workload\,destination_ip\,
      destination_namespace\,destination_workload\,traffic_direction,tcp:labelsContext=source_ip\,
      source_namespace\,source_workload\,destination_ip\,destination_namespace\,destination_workload\,
      traffic_direction}" \
  --set hubble.eventQueueSize=65535 \
  --set resources.requests.memory="1000Mi" \
  --set resources.requests.cpu="2000m" \
  --set rollOutCiliumPods="true" \
  --set devices[0]="ens3" \
  --set ipam.operator.clusterPoolIPv4PodCIDRList[0]="192.168.0.0/16" \
  --set-string bpf.monitorAggregation="medium" \
  --set nodePort.enabled=true \
  --set ingressController.enabled=true \
  --set ingressController.loadbalancerMode=dedicated \
  --set envoyConfig.enabled=true
```

E.2 Application Stack

Listing 9: Deploying OpenEBS and the Train-Ticket workload

```
# Untaint control-plane nodes so OpenEBS workloads can schedule
CONTROL_PLANES=$(kubectl get nodes | grep control-plane | awk '{print $1}')
kubectl taint nodes $CONTROL_PLANES node-role.kubernetes.io/master:NoSchedule-
kubectl taint nodes $CONTROL_PLANES node-role.kubernetes.io/control-plane:NoSchedule-

# Storage layer (OpenEBS)
kubectl apply -f https://openebs.github.io/charts/openebs-operator.yaml
# Wait for pods, re-taint control-plane nodes
kubectl taint nodes $CONTROL_PLANES node-role.kubernetes.io/master:NoSchedule
kubectl taint nodes $CONTROL_PLANES node-role.kubernetes.io/control-plane:NoSchedule

# Make OpenEBS default StorageClass
kubectl patch storageclass openebs-hostpath \
  -p '{"metadata":{"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
# Make Cinder (StorageClass of the used cloud provider) as non-default
kubectl patch storageclass cinder-storageclass \
  -p '{"metadata":{"annotations":{"storageclass.kubernetes.io/is-default-class":"false"}}}'

# Train-Ticket workload
git clone https://github.com/FudanSELab/train-ticket.git
cd train-ticket
make deploy DeployArgs="--all"

# Update service ports
## List of services to be updated
services=(
    "ts-admin-basic-info-service"
    "ts-admin-order-service"
    "ts-admin-route-service"
    "ts-admin-travel-service"
    "ts-admin-user-service"
    "ts-assurance-service"
    "ts-auth-service"
    "ts-avatar-service"
    "ts-basic-service"
    "ts-cancel-service"
    "ts-config-service"
    "ts-consign-price-service"
    "ts-consign-service"
    "ts-contacts-service"
    "ts-delivery-service"
    "ts-execute-service"
    "ts-food-service"
    "ts-gateway-service"
    "ts-inside-payment-service"
    "ts-news-service"
    "ts-notification-service"
    "ts-order-other-service"
    "ts-order-service"
    "ts-payment-service"
    "ts-preserve-other-service"
    "ts-preserve-service"
    "ts-price-service"
    "ts-rebook-service"
    "ts-route-plan-service"
    "ts-route-service"
    "ts-seat-service"
```

```
    "ts-security-service"
    "ts-station-food-service"
    "ts-station-service"
    "ts-ticket-office-service"
    "ts-train-food-service"
    "ts-train-service"
    "ts-travel-plan-service"
    "ts-travel-service"
    "ts-travel2-service"
    "ts-ui-dashboard"
    "ts-user-service"
    "ts-verification-code-service"
    "ts-voucher-service"
)
# Iterate over the services and patch each one
for service in "${services[@]}"
do
    # Get the current ports information
    ports=$(kubectl get service $service -o json | jq -r '.spec.ports')

    # Iterate over each port and create a new ports array with port set to 80
    new_ports=$(echo $ports | jq 'map(.port = 80)')

    # Patch the service with the updated ports array
    kubectl patch service $service --type merge -p "{\"spec\":{\"ports\": $new_ports}}"
done

# Update service resources

## Set the desired resource requests and limits
CPU_REQUEST="100m"
MEMORY_REQUEST="100Mi"
CPU_LIMIT="1000m"
MEMORY_LIMIT="700Mi"
## Update resource requests and limits for the specified deployments
for service in "${services[@]}"; do
  kubectl get deployment "$service" -n default -o json | jq --arg cpu_req "$CPU_REQUEST" --arg mem_req "
      $MEMORY_REQUEST" --arg cpu_lim "$CPU_LIMIT" --arg mem_lim "$MEMORY_LIMIT" '
  .spec.template.spec.containers[] |=
    (.resources.requests.cpu = $cpu_req |
     .resources.requests.memory = $mem_req |
     .resources.limits.cpu = $cpu_lim |
     .resources.limits.memory = $mem_lim) |
  del(.metadata.creationTimestamp, .status) |
  {"namespace": .metadata.namespace, "name": .metadata.name, "deployment": .}
  ' | jq -c '.deployment' | kubectl apply -f -
  echo "Updated resources for deployment $service in namespace default"
done

# Patch Java containers to reduce JVM footprint
kubectl get deployments -o json | jq -c '
 .items[]
 | select(.spec.template.spec.containers[].env[]? | select(.name=="JAVA_TOOL_OPTIONS"))
 | {name: .metadata.name,
    namespace: .metadata.namespace,
    containers: [
      .spec.template.spec.containers[]
      | select(.env[]?.name=="JAVA_TOOL_OPTIONS")
      | {name: .name,
         envIndex: [.env | to_entries | .[] | select(.value.name=="JAVA_TOOL_OPTIONS").key][0]}
    ]}'
| while read -r deployment; do
```

```
  name=$(echo "$deployment" | jq -r '.name')
  namespace=$(echo "$deployment" | jq -r '.namespace')
  for container in $(echo "$deployment" | jq -c '.containers[]'); do
      containerName=$(echo "$container" | jq -r '.name')
      envIndex=$(echo "$container" | jq -r '.envIndex')
      current=$(kubectl get deployment "$name" -n "$namespace" -o \
        jsonpath="{.spec.template.spec.containers[?(@.name==\"$containerName\")].env[$envIndex].value}")
      new="${current} -Xshare:auto -XX:TieredStopAtLevel=1 -XX:CICompilerCount=1 \
           -XX:+UseSerialGC -XX:-UsePerfData -Dhttp.keepAlive=false"
      kubectl patch deployment "$name" -n "$namespace" --type=json \
        -p="[{'op':'replace','path':\"/spec/template/spec/containers/0/env/$envIndex/value\",'value':\"$new
            \"}]"
  done
done
```

E.3 Applying Cilium Layer 7 Network Policy (CNP)

Listing 10: Train-Ticket CNP

```
kubectl apply -f - <<'EOF'
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "l7-visibility"
spec:
  endpointSelector:
    matchLabels:
      "k8s:io.kubernetes.pod.namespace": default
  ingress:
  - toPorts:
    - ports:
      - port: "17853"
        protocol: TCP
      - port: "12032"
        protocol: TCP
      - port: "12031"
        protocol: TCP
      - port: "19001"
        protocol: TCP
      - port: "14569"
        protocol: TCP
      - port: "14568"
        protocol: TCP
      - port: "16579"
        protocol: TCP
      - port: "18886"
        protocol: TCP
      - port: "14578"
        protocol: TCP
      - port: "11178"
        protocol: TCP
      - port: "18898"
        protocol: TCP
      - port: "11188"
        protocol: TCP
      - port: "18855"
        protocol: TCP
      - port: "12345"
```

```
      protocol: TCP
    - port: "16108"
      protocol: TCP
    - port: "19999"
      protocol: TCP
    - port: "14567"
      protocol: TCP
    - port: "14322"
      protocol: TCP
    - port: "12346"
      protocol: TCP
    - port: "16346"
      protocol: TCP
    - port: "12342"
      protocol: TCP
    - port: "15678"
      protocol: TCP
    - port: "16101"
      protocol: TCP
    - port: "32677"
      protocol: TCP
    - port: "80"
      protocol: TCP
    rules:
      http: [{}]
- toPorts:
  - ports:
    - port: "80"
      protocol: TCP
    - port: "9200"
      protocol: TCP
    - port: "9300"
      protocol: TCP
    - port: "443"
      protocol: TCP
    - port: "8848"
      protocol: TCP
    - port: "7848"
      protocol: TCP
    - port: "5672"
      protocol: TCP
    - port: "12800"
      protocol: TCP
    - port: "11800"
      protocol: TCP
    - port: "8080"
      protocol: TCP
    - port: "18767"
      protocol: TCP
    - port: "16112"
      protocol: TCP
    - port: "16113"
      protocol: TCP
    - port: "16114"
      protocol: TCP
    - port: "16115"
      protocol: TCP
    - port: "18888"
      protocol: TCP
    - port: "12340"
      protocol: TCP
    - port: "17001"
```

```
        protocol: TCP
      - port: "15680"
        protocol: TCP
      - port: "18885"
        protocol: TCP
      - port: "15679"
        protocol: TCP
      - port: "16110"
        protocol: TCP
      - port: "16111"
        protocol: TCP
      - port: "12347"
        protocol: TCP
      - port: "18808"
        protocol: TCP
      - port: "12386"
        protocol: TCP
      - port: "18856"
        protocol: TCP
      - port: "18673"
        protocol: TCP
      - port: "12862"
        protocol: TCP
      rules:
        http: [{}]
  - fromEndpoints:
    - matchLabels:
        "k8s:io.kubernetes.pod.namespace": default
    icmps: []
  egress:
  - toPorts:
    - ports:
      - port: "12800"
        protocol: ANY
      - port: "8848"
        protocol: ANY
      - port: "53"
        protocol: UDP
      - port: "3306"
        protocol: ANY
      rules: {}
  - toEntities:
    - all
    # Allow all ICMP traffic
  - icmps:
    - {}
EOF
```

E.4 Monitoring — Prometheus Stack

Listing 11: Prometheus + Grafana stack

```
# Remove any current prometheus installation, for the used provider was in the namespace mgk-monitoring
kubectl delete ns mgk-monitoring
# Install Prometheus+Grafana stack - may need custom configurations depending on the cloud provider
helm install prometheus prometheus-community/kube-prometheus-stack \
  --namespace prometheus --create-namespace
```

E.5 Installing `metrics-parser-api` and `dependency-calculator`

Listing 12: Installing metrics-parser-api and dependency-calculator

```
# metrics_parser_api.yaml should have the content of the metrics-parser-api deployment file earlier
    presented
kubectl create -f metrics_parser_api.yaml
# dependency_calculator.yaml should have the content of the dependency-calculator deployment file earlier
    presented
kubectl create -f dependency_calculator.yaml
```