Mesh-Aware Debugging: Identifying Resource Allocation Issues in Distributed Microservices

Márk Emődi[1,2,*], József Kovács[1] and Róbert Lovas[1,2]

[1]HUN-REN Institute for Computer Science and Control (HUN-REN SZTAKI), Hungarian Research Network, Kende utca 13-17, H-1111 Budapest, Hungary; {mark.emodi, jozsef.kovacs, robert.lovas}@sztaki.hun-ren.hu

[2]John von Neumann Faculty of Informatics, Óbuda University, Bécsi út 96/b, H-1034 Budapest, Hungary; {emodi.mark, lovas.robert}@nik.uni-obuda.hu

* Corresponding author

Abstract: In microservice architectures, a core operational challenge lies in effectively monitoring the system. This involves aggregating diverse data types including logs, metrics, traces, events, and topology from every service and its individual instances. To tackle these issues, an experimental framework is introduced that applies macrostep debugging principles within a service mesh environment. This novel approach enables more controlled and systematic analysis of microservice communication and operation. The effectiveness of this system is validated through a resource allocation problem, showcasing its ability to detect and actively control an application in a failure state. Furthermore, a performance evaluation demonstrates that although the debugging framework introduces additional overhead in terms of latency and throughput, it remains effective for analyzing and reproducing faulty states in distributed environments. These results confirm that the approach provides valuable debugging insights at the cost of moderate but manageable performance trade-offs.

Keywords: Service mesh; Debugging; Distributed systems; Istio; Kubernetes; Resource allocation

1 Introduction

The challenge of debugging a microservice environment is significant and induced by the complexity and distribution inherent in such systems. In monolithic architectures, it is easier to analyze the entire application as a single unit than to work with multiple small components that are independently deployed and may make network calls to interact with each other. This increases the degree of difficulty in detecting faults.

One of the main issues is the magnitude of logs produced by microservices. Developers must frequently sift through vast amounts of log data to identify faults [1], a time-consuming activity that requires extensive developer knowledge. This is especially problematic, in large systems that can produce millions of traces per second, which makes it difficult to distinguish real signals from the noise [2]. A large number of trace data may also slow down the process of fault analysis and debugging.

Moreover, the dynamic nature of microservice interactions introduces additional layers of challenge. Each microservice may communicate with numerous other services, resulting in complex call chains that make it difficult to pinpoint the source of a failure [3] [4]. The decentralized nature of microservice architecture also implies that issues can stem from various origins, including hardware failures, implementation flaws, or even incorrect service coordination.

Further complicating matters are technological aspects relating to observability and instrumentation. Effective debugging relies on robust observability measures, yet instrumentation can introduce performance overheads that complicate the debugging process [5]. Developers often struggle with balancing the need for detailed monitoring with the overhead induced by gathering such metrics [6]. The result is a landscape where the efficiency of debugging tools is crucial, yet often insufficient.

Additionally, as highlighted in [7], integration challenges and configuration errors are prevalent issues in microservice environments, which further complicates debugging efforts. This multifaceted landscape requires a systematic approach to analyze potential faults, emphasizing the need for sophisticated tools capable of managing distributed systems effectively.

In summary, debugging microservice architectures is inherently complex due to high log volume, complex inter-service interactions, and the challenge of achieving deep observability without incurring significant performance trade-offs. Addressing these challenges requires not only sophisticated tracing and monitoring techniques but also robust organizational strategies to effectively manage the system's complexity. In this paper, the challenges are explained in detail and a possible solution is proposed to enhance the reliability of microservice debugging combining previous debugging methods with this architectural approach. The paper presents a structured approach to detailing the methodology and demonstrates the proposed solution.

This work is structured as follows. Section 2 reviews related work in microservice debugging, examining existing approaches for fault detection and their limitations in distributed environments. Section 3 presents our methodology and introducing macrostep-based debugging principles adapted for service mesh architectures. Section 4 outlines the overall concept of our experimental framework, describing the integration of WebAssembly filters and active control mechanisms. Section 5 details the operational flow of our approach, illustrating how network requests are

intercepted, controlled, and analyzed within the service mesh environment. Section 6 evaluates the performance impact of our debugging framework through comprehensive testing and analysis. Finally, Section 7, concludes the paper with a summary of contributions and directions for future work.

2 Related Work

Microservice architectures have transformed the development and deployment of software applications, promoting scalability and modularity. However, they also introduce substantial challenges in debugging due to their inherent complexity. In a microservice system, each node deploys one or more containers that are grouped into pods. Each service may have multiple instances, and services communicate with each other over the network. Zhang et al. [8] categorize failure diagnosis into three levels:(i) Service level, (ii) Instance level, and (iii) Component level. Service-level and instance-level detection help identify which service or which instance within a service is the root cause of a failure. However, component-level localization offers a more granular diagnosis by not only identifying the problematic service or instance but also identifying the specific component within it. The effectiveness of this granularity largely depends on whether the solution handles the service or instance as a single unit or examines its internal structure.

Yu et al. [9] introduced a tracing-based approach designed to identify root causes of latency issues in microservice environments. Their system extracts service latency information from tracing data and applies anomaly detection techniques. However, if latency is not a factor in the fault condition, the method may fail to identify the underlying issue.

Wu et al. [10] propose a solution that utilize service dependency graphs to model service interactions, enabling the identification of faulty services based on their relationships. Their approach constructs causality graphs that represent both communicating and non-communicating dependencies, which are critical for tracing fault propagation paths. The culprit metric localization method is constrained to detecting root causes that exhibit noticeable deviations from expected or normal values.

One prominent method for debugging in microservices is delta debugging, as proposed by Zhou et al. [1]. They suggest representing microservice system settings as different scenarios and applying delta debugging to isolate failure-inducing conditions within these contexts. This approach simplifies the process by allowing developers to identify the minimal change required to reproduce a fault, facilitating faster diagnosis and rectification of issues within microservices. Limitations of this method can arise when applying this method in larger microservices networks, as the interactions between services complicate the detection of failure causes.

Liu et al. [11] explore anomaly propagation via dynamic service call graphs, ranking potential root causes with a combination of machine learning and statistical methods. Demonstrated substantial efficiency improvements. Their approach demonstrated significant efficiency gains, reducing localization time from 30 minutes to 5 minutes. However, its reliance on comprehensive multi-source data collection infrastructure in cloud-edge environments and the computational overhead of maintaining heterogeneous dynamic topology stacks, which may limit scalability and efficiency in large-scale or frequently changing topologies.

Lee et al. [12] utilizes trace logs and ML models to predict latent errors, pinpoint faulty microservices, and identify fault types with high precision and recall in benchmarks, such as Sock Shop and Train Ticket. The main limitation of their solution is its heavy dependency on comprehensive multi-source data collection infrastructure and substantial labeled training data, making it challenging to deploy in real production environments where such resources may not be readily available.

3 Methodology

3.1 Debugging Across System Phases

During debugging, it is important to distinguish between two types of phases: (i) the deployment phase, where the application begins its operation, and (ii) the operational (runtime) phase, when it is running in a production environment.

In the deployment phase, the intervention toolkit is significantly broader. In this state, there are more opportunities to observe and analyze the application's behavior in detail. Various diagnostic tools, such as debuggers, profilers, extended logging levels, and trace and metrics collection, can be more easily applied or injected. Using an agent also offers greater flexibility, as it is not constrained by the strict performance, security, or availability requirements of a live environment. As a result, the identification of issues and the analysis of cause-and-effect relationships can be performed more efficiently and quickly.

In contrast, during the operational phase, debugging is carried out with a much more limited toolkit. At this stage, the application is already running, and any disruption of service, decrease in performance, or damage to data integrity can have severe consequences. Therefore, intervention options are minimized, and most monitoring capabilities must be provided in advance (e.g., proper logging, integration of monitoring systems), so that any potential issues can be traced afterward.

Our research focused on the system's operational phase, during which the components are already running and active communication takes place between them. In this operational, stable state, examining network traffic is particularly relevant, as the observation has a smaller impact on the system's performance.

The data packets and protocols operate according to the normal behavior of the system, enabling clear identification of communication patterns. This creates an opportunity for effectively detecting and interpreting any anomalies or irregularities.

3.2 Macrostep-based Debugging

Parallel and distributed systems create unique challenges in debugging due to their inherently non-deterministic nature. Variability in runtime conditions, such as differing CPU/memory speeds, operating system scheduling, and network latencies, can cause the same program to behave differently across executions. This non-determinism significantly complicates the debugging process, particularly when attempting to reproduce erroneous behavior.

Traditional sequential debugging methodologies, which rely heavily on deterministic execution and stepwise control using breakpoints, are less suitable for such environments. Early approaches to debugging parallel systems often used the "monitor and replay" technique [13]. During the monitoring (or recording) phase, the tool captures information about the parallel program to enable a deterministic reproduction of the execution. In the subsequent replay phase, the program's execution is reproduced based on the previously collected data. While this approach offers a way to debug parallel programs, it introduces a new issue known as the probe effect. This occurs when the act of monitoring alters the program's timing behavior. Although reducing the amount of data collected during monitoring can lessen the impact, the probe effect itself cannot be fully eliminated.

Another method for debugging parallel programs is the "control and replay" technique (also called active control) [13] [14]. In this approach, systematically generated test cases are used to exhaustively cover all possible timing conditions in the program. Replay is driven not by previously collected data but by these generated test cases. The central challenge of this method lies in developing an effective way to generate such test cases.

To overcome the monitor and replay limitations, macrostep debugging introduces a comprehensive, reproducible, and systematic approach for identifying and analyzing faults in message-passing programs. It combines the rigor of formal execution models with the practicality of breakpoint-based debugging, extending well-understood sequential techniques into the more complex domain of concurrent systems. In macrostep debugging, test cases are actively generated to steer program execution. These test cases are constructed systematically to exhaustively explore all possible timing conditions and execution sequences. The replay phase is driven by these test cases, enabling exhaustive and reproducible coverage of the program's non-deterministic behavior. A central controller orchestrates execution, enforcing specific timing scenarios and coordinating process states in accordance with the

generated test cases. This approach ensures that all possible overlaps and race conditions can be systematically evaluated.

Macrostep debugging is built around several fundamental concepts. It distinguishes between the *local* level and the *global* level. Local breakpoints operate at the individual process level and halt execution when a predefined program state is reached. A collective breakpoint consists of a coordinated set of local breakpoints, ideally including one from each process involved in the computation. A collective breakpoint is said to be *complete* if it contains a local breakpoint from every active process and *strongly complete* if it accounts for all alternative execution paths across processes, ensuring comprehensive coverage of concurrency scenarios.

Execution between two collective breakpoints defines a macrostep. A macrostep represents a semantically meaningful unit of execution that abstracts away fine-grained thread or message-level interactions. Macrosteps are further classified based on the nature of their operations. A *pure* macrostep is one in which communication-related operations occur only at the end of the step, thereby preserving clean boundaries for analysis. The original macrostep-debugging concept differentiates between *pure* and *compound* macrosteps. A macrostep is considered *pure* if communication-related code appears only at its final element; otherwise, it is classified as *compound*. By requiring macrosteps to be pure and collective breakpoints to be strongly complete, macrostep debugging ensures a reliable approach. This allows the traditional breakpoint-to-breakpoint debugging method, commonly used in sequential programs, to be effectively extended to parallel execution. This facilitates macrostep-by-macrostep debugging, offering a clearer understanding of concurrent behavior and enabling step-by-step reasoning about distributed system execution.

To manage the combinatorial complexity of parallel execution paths, macrostep debugging introduces the concept of an execution tree. This tree encapsulates all feasible execution paths and timing conditions in a program. In this structure, nodes represent collective breakpoints, while edges denote macrosteps. The root node marks the program's starting point, and each path through the tree corresponds to a specific sequence of collective breakpoint activations, or an execution path. The tree incorporates three distinct types of nodes:

1. **Deterministic nodes**, which transition to exactly one next state without branching into multiple execution paths,
2. **Alternative nodes**, which represent mutually exclusive decisions that lead to different outcomes depending on the timing of events or the order of message receptions,
3. **Fork nodes**, which introduce true concurrency by allowing the program to branch into multiple parallel threads of execution.

This hierarchical representation allows developers to systematically navigate, visualize, and manipulate the program's state space. Furthermore, meta-breakpoints

can be strategically placed within the execution tree to steer the program toward specific states or to force the evaluation of certain timing scenarios. These meta-breakpoints act as high-level control constructs, enabling selective exploration of the execution tree without the need to exhaustively traverse all paths in each debugging session.

The primary goal of macrostep debugging is to expose and analyze concurrency-induced faults by discovering different timing combinations among concurrent operations in distributed environments. This technique enables a controlled and comprehensive exploration of execution paths. As a result, it supports the reproducible detection of difficult-to-trace errors such as race conditions and deadlocks. Unlike ad hoc testing or trace-based analysis, macrostep debugging provides a structured method in which every potential execution order can be systematically tested and verified.

3.2 Macrostep-based Debugging in Service Mesh

Figure 1 illustrates a complex synthetic example designed for demonstration purposes, featuring several collective breakpoints such as $NAR^1{}_1$–$NAS^1{}_2$–$NAS^1{}_3$–$NAR^1{}_4$. These breakpoints are positioned on inter-MS (Microservice) communication primitives related to the sender (NAS), receiver (NAR), or alternative/collective receiver (NACR) methods within each MS communication process. Inter-MS communications are typically synchronized actions, indexed by the corresponding MS communication process number (subscript) and a serial number (superscript), as shown in Figure 1. The series of upper-level communication steps executed between two consecutive collective breakpoints is referred to as a macrostep.

A single breakpoint within a collective breakpoint is considered active if it is triggered during a macrostep and its associated inter-MS communication completes successfully (e.g., $NAS^2{}_2$ in Figure 1). Conversely, a breakpoint is classified as sleeping if it is triggered but its corresponding notification cannot be completed in the current macrostep, thus carrying over to the next collective breakpoint.

For example, consider a send instruction ($NAS^2{}_1$) from one MS process (MS1) attempting synchronous communication with another process (MS4). If MS4 is already engaged with a third MS process (MS3), the breakpoint at $NAS^2{}_1$ becomes a sleeping breakpoint and will appear in the subsequent collective breakpoint. Similarly, $NAS^3{}_4$ is also a sleeping breakpoint, as it must wait for $NAS^4{}_5$ to proceed.

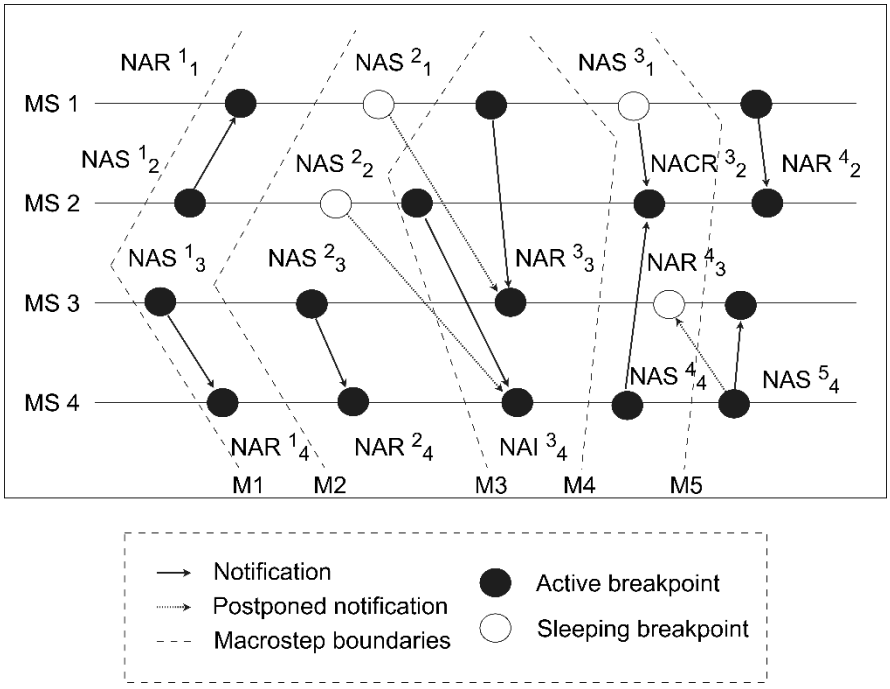

Illustration for the macrostep-based execution of complex deployment with MS based communication

Microservice architectures can involve numerous services, frequently deployed and scaled, making it difficult to analyze errors due to the non-deterministic nature of cloud resources. Factors, such as varying service configurations, diverse dependencies, and fluctuations in resource load (CPU, memory, network) can all affect timing during communication and operation. This complexity often leads to situations where errors are hard to reproduce consistently.

Istio [15] built on the top of Kubernetes delivers sophisticated traffic management capabilities and precise control over service-to-service interactions. We built our solution by leveraging the flexibility of the network layer. Through its integration with Kubernetes' native functionalities, Istio strengthens the network layer, scalability, and maintainability of distributed applications in cloud-native ecosystems.

Applying the macrostep debugging methodology to microservice orchestration is promising. In this context, the "processes" are the individual microservices. By inserting local breakpoints within each microservice's network communication, a debugger can pause network communication for analysis and control.

These local breakpoints can be grouped into collective breakpoints, where each collective breakpoint represents a synchronized pause across all relevant microservices. Reaching a collective breakpoint signifies a consistent global state

of the microservice system, allowing for inspection of each service's state. If the debugger has a choice of which microservice to allow to proceed, it's an alternative collective breakpoint. If there is only one service that can continue, then the breakpoint is deterministic.

The execution tree represents all possible execution paths and timing scenarios during the microservice communication and operation. The root node of this tree is the set of initial local breakpoints for each microservice. Connections between nodes are macrosteps, representing the execution between two collective breakpoints. This approach enables debugging of microservice network and operation step-by-step, moving from one collective breakpoint to the next.

4 Overall Concept

We present an experimental framework that leverages a WebAssembly (WASM) filter and Python to implement an active control mechanism for network requests within a controlled test environment. The framework consists of multiple components that integrate with the service mesh, facilitating the evaluation of various use cases. To extend the capabilities of the service mesh experiment system, we designed and incorporated additional components that enable precise traffic manipulation within the architecture. By leveraging the extended network layer of the service mesh, our approach introduces custom components that support traffic blocking and order manipulation, which enhances the flexibility and functionality of the proposed architecture.

An example application was created to demonstrate the benefits of macrostep debugging. It is based on a resource allocation problem, the dining philosophers [16]. The dining philosopher's problem is a classical synchronization problem that models resource allocation in concurrent computing. Reflecting on this problem, it becomes indisputable that efficient resource allocation is critical to ensuring that philosophers can utilize resources ("eat") without causing deadlocks or resource starvation. Deadlock happens when each philosopher picks up one fork and then waits for the other. This creates a circular wait where no one can continue, so all progress stops.

The application is built on microservice architecture, where distinct services and interfaces communicate with each other to manage resource allocation. Each service allocates resources and releases them after a randomized duration. The core philosophy of the application is to simulate resource consumption based on real-life use cases.

In certain scenarios, a service may require multiple resources simultaneously. If components behave greedily, demanding additional resources to complete tasks (such as calculations), the risk of potential deadlocks could increase. This risk is

further amplified by increasing the number of components competing for resources and reducing the time spent on computation, leading to more frequent resource allocation attempts.

The demo application demonstrates this behavior described above and is written in Go. It consists of six processes and six resources, with each process requiring exactly two resources to operate. Resource allocation is determined at runtime, making the order of allocation significant. Although this configuration typically functions without issues, deadlock occurs when all processes simultaneously hold exactly one resource while waiting indefinitely for their second required resource.

Our solution successfully identified the resource allocation issue in the application based on the monitored states and actively guided toward the faulty condition at runtime through active control. Beyond the experimental dining philosopher's application, the proposed framework is applicable to other real-world microservice scenarios where concurrency and timing issues frequently arise. To mitigate this, various algorithms and frameworks from cloud computing, notably those that utilize dynamic resource allocation strategies, provide valuable insight. For example, in cloud-based job scheduling on platforms like Kubernetes or SLURM, multiple services may race for CPU or GPU resources, and subtle race conditions in allocation may cause starvation or indefinite waiting. These scenarios highlight how systematic, macrostep-based debugging in a service mesh can move beyond theory by uncovering and reproducing resource contention, race conditions, and deadlocks in distributed environments.

5 Operational Flow

Figure 2 illustrates a microservices-based application consisting of two independent services that communicate with each other. In a service mesh environment, this communication is handled by sidecar components (such as the Envoy Proxy), which are responsible for managing the networking layer. The sidecars discover destination addresses and route traffic to the designated services. The proxy layer also offers various capabilities to influence traffic behavior, including monitoring, controlling, and mirroring traffic. To enable traffic monitoring and control, an external component has been developed, making use of the sidecar's extensibility.

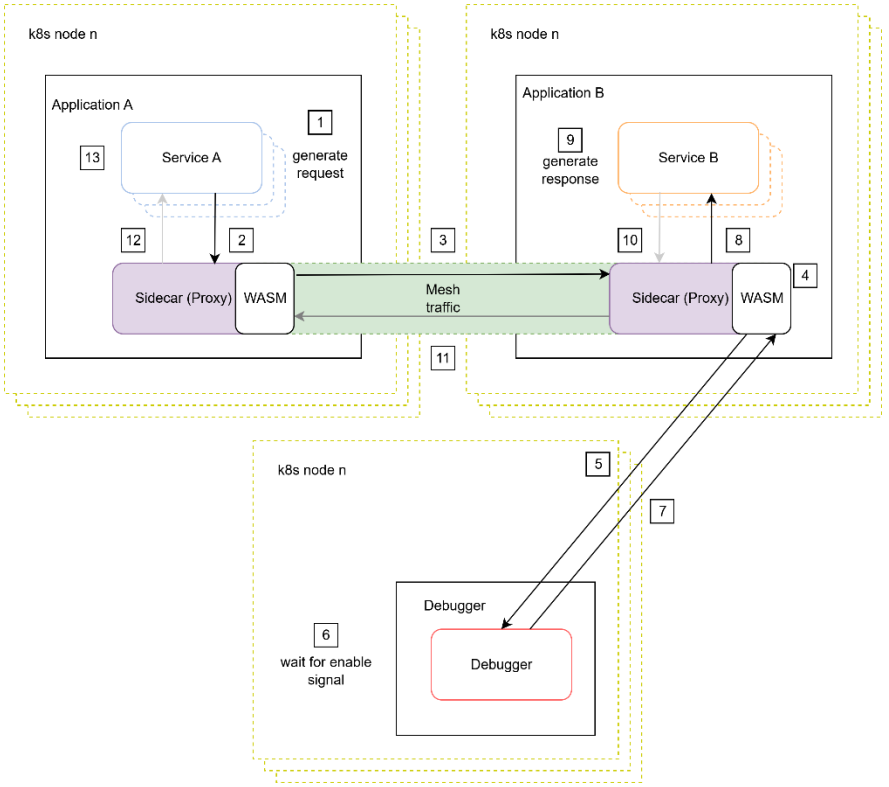

Figure 2
The proposed architecture outlines the flow of network requests

When Application A initiates communication with Application B, the message exchange proceeds through the following steps:

1. **Message Preparation and Transmission:** Application A prepares the message request and sends it to the destination. (Step 1)
2. **Interception by Sidecar:** The request is intercepted by the local sidecar proxy (e.g., Envoy), which operates transparently alongside the application. (Step 2)
3. **Forwarding to Destination Sidecar:** The message is forwarded to the sidecar proxy of Application B. (Step 3)
4. **Custom Processing via WASM Filter:** The receiving sidecar is extended with a WASM filter (implemented using the Go SDK), which enables the injection of custom logic. In our implementation, this filter notifies a controller component about the incoming message, logs the network request, and actively control its delivery. (Step 4)

5. **Conditional Delivery:** Until the controller approves the connection, message delivery is delayed. To accommodate this, timeouts should be disabled on the application side to prevent timeout request termination. (Step 7)
6. **Acknowledgment and Final Delivery:** Once approved, an acknowledgment is sent, and the proxy delivers the message to Application B. (Step 8)
7. **Response Handling:** Application B processes the message and generates a response, which is intercepted by its sidecar, forwarded to Application A's sidecar, and finally delivered to Application A. (Step 9-13)

Based on the methodology described above, the system was utilized to address a resource allocation problem, specifically the Dining Philosophers scenario. The faulty state was successfully identified by our solution through the mapping of system states, taking into account external timing parameters. Furthermore, through active control, the system is capable of steering the application toward this state for further evaluation.

6 Performance Evaluation

When considering a debugging tool, evaluating its performance is essential, since the speed and effectiveness of identifying and resolving issues depend directly on how well the tool performs. A reliable tool ensures accurate results, supports efficient workflows, and contributes to overall software quality instead of creating obstacles.

Table 1
Performance comparison of Controlled and Normal services

Metrics	**Normal service**	**Controlled service**
Avg Latency (ms)	1.63	16.81
Latency Stdev (ms)	0.95	80.62
Total Requests/sec	4958.00	1120.37
Requests (30s)	148818	33649

However, it is important to highlight the limitations of the current implementation. At present, the component represents a single point of failure within the system. If a failure occurs, Kubernetes is expected to redeploy the application. In the case of a node-level failure, the redeployment might occur on a different node. When using distributed storage solutions (e.g., Rook), this process should ensure that no data is lost.

The impact on the network was evaluated using the WRK tool, and the measurements were repeated five times and the average was represented. In our measurements, we ignored node-level scheduling and instead allowed Kubernetes to manage component scheduling. This approach lets Kubernetes operate as it would in normal scenarios, making placement decisions based on resource availability and policies. As a result, the measurements more accurately represent ordinary behavior, reduce bias from manual scheduling, and provide a clearer evaluation of system performance.

Table 1 presents a comparison of system performance under normal and controlled service conditions. Under normal service, the average latency was 1.63 ms with a standard deviation of 0.95 ms, while under controlled service, the average latency increased significantly to 16.81 ms with a much higher standard deviation of 80.62 ms, indicating greater variability in response times. The total throughput also decreased substantially under controlled service, with the system handling 4958 requests per second in normal service versus 1120.37 requests per second in controlled service. Over a 30-second interval, this corresponds to 148818 requests processed under normal conditions compared to 33649 requests under controlled conditions, demonstrating a marked reduction in processing capacity. Evaluating the network metrics, the source of the additional overhead is that during control and monitoring, an extra packet is created for the controller, which adds extra network latency and decrease the throughput of the system. Since the examined application did not contain significant business logic and was written in go, very low latency values were achieved during normal operation. Therefore, it can be assumed that in a real-world usage scenario with multiple microservices, the latency of the normal service would be higher, while the latency of the controlled service would remain unchanged. In our future work, we plan to conduct a more detailed analysis of the performance aspect.

Figure 3 illustrates the latency distribution for the normal and controlled service across selected percentiles (P50, P75, P90, and P99). The results show that the normal service maintains consistently low latency values, ranging from 1.47 ms at the 50^{th} percentile to 3.49 ms at the 99^{th} percentile. In contrast, the controlled service exhibits noticeably higher latency across all percentiles, with values of 6.96 ms (P50), 10.15 ms (P75), and 12.75 ms (P90). A significant divergence is observed at P99. These results indicate that while the overhead introduced by the control mechanism is moderate under typical conditions, it can lead to substantial latency outliers in extreme cases.

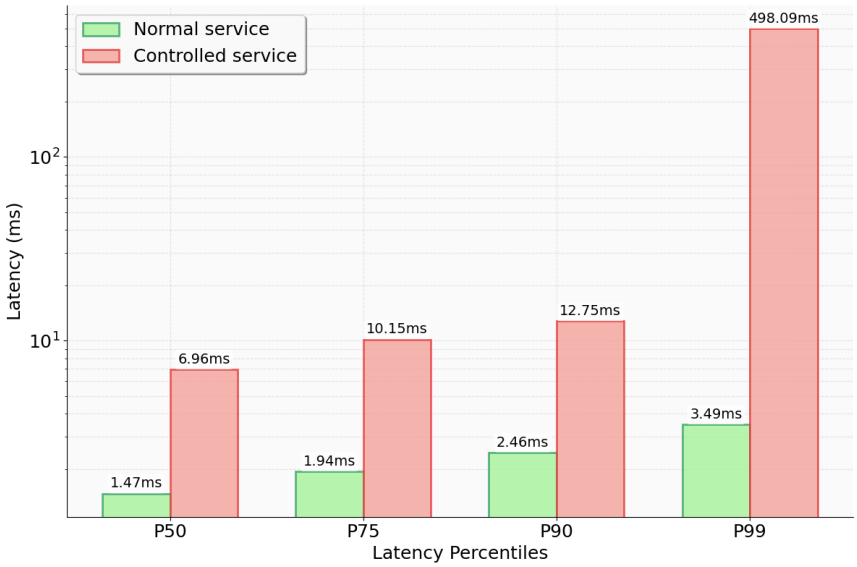

Figure 3
Latency comparison between normal and controlled services

Conclusions

In this paper, we have addressed the key challenges associated with debugging microservice architectures, emphasizing the complexities arising from distributed systems, high log volumes and intricate service interactions.

We designed an experimental framework that leverages macrostep debugging principles within a service mesh environment, for error detection, with active monitoring and controlling. We devised and realized an experimental framework that leverages macrostep debugging principles within a service mesh environment for error detection with active monitoring and controlling. In this resource allocation scenario, we successfully achieved error detection by actively monitoring and controlling microservice interactions. Through the integration of multiple components, our system was able to intercept and manipulate network requests, enabling us to identify faulty states that would otherwise be difficult to detect using conventional debugging methods. The approach proved valuable in a dynamic and timing-sensitive environment like the Dining Philosophers problem, where time conditions and resource allocations are critical. In addition, a performance evaluation was carried out, which showed that the framework introduces additional latency and reduces throughput compared to normal operation.

Our current implementation, necessitates disabling application-side timeouts, to accommodate debugging delays, which introduces a trade-off that requires careful management of potential indefinite wait scenarios. To address this limitation, we are exploring alternative approaches, such as application-side instrumentation.

We demonstrated the effectiveness of our system on a resource allocation problem by successfully detecting and actively controlling the application in a failure state. This approach enables a more controlled and systematic analysis of microservice communication and operation, as demonstrated by our implementation using the Dining Philosophers problem. By integrating a WASM filter and utilizing a controller for active network request control, our framework offers a promising way of enhancing the reliability and debuggability of microservice-based applications.

Future work will aim to refine the framework further, investigate its applicability to a wider variety of complex microservice scenarios and provide a more comprehensive performance evaluation.

Acknowledgements

Project no. 149909 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the MEC_R funding scheme. The authors thankfully acknowledge the support of the Doctoral School of Applied Informatics and Applied Mathematics, Óbuda University. On behalf of the ”Modelling of orchestration methods with machine learning” project, we are grateful for the possibility of using HUN-REN Cloud (see [17]; https://science-cloud.hu/), which helped us achieve the results published in this paper.

References

[1] X. Zhou *et al.*, “Delta Debugging Microservice Systems with Parallel Optimization,” *IEEE Trans. Serv. Comput.*, Vol. 15, No. 1, pp. 16-29, Jan. 2022, doi: 10.1109/TSC.2019.2919823

[2] B. Li *et al.*, “Enjoy your observability: an industrial survey of microservice tracing and analysis,” *Empir. Softw. Eng.*, Vol. 27, No. 1, p. 25, Jan. 2022, doi: 10.1007/s10664-021-10063-9

[3] M. Ekhlasi, F. F. Daneshgar, M. Dagenais, M. Lamothe, N. Ezzati-Jivan, and M. Khouzam, “DTraComp: Comparing distributed execution traces for understanding intermittent latency sources,” Oct. 18, 2023, *Preprints*. doi: 10.22541/au.169762695.52482914/v1

[4] Z. H. Zhiqiang Hao, X. Z. Zhiqiang Hao, J. L. Xufan Zhang, and Q. W. Jia Liu, “Service Call Chain Analysis for Microservice Systems,” *J. Internet Technol.*, Vol. 23, No. 6, pp. 1203-1211, Nov. 2022, doi: 10.53106/160792642022112306004

[5] H. M. Kabamba, M. Khouzam, and M. R. Dagenais, “Vnode: Low-Overhead Transparent Tracing of Node.js-Based Microservice Architectures,” *Future Internet*, Vol. 16, No. 1, p. 13, Dec. 2023, doi: 10.3390/fi16010013

[6] L. Wu, J. Bogatinovski, S. Nedelkoski, J. Tordsson, and O. Kao, “Performance Diagnosis in Cloud Microservices Using Deep Learning,” in *Service-Oriented Computing – ICSOC 2020 Workshops*, Vol. 12632, H.

Hacid, F. Outay, H. Paik, A. Alloum, M. Petrocchi, M. R. Bouadjenek, A. Beheshti, X. Liu, and A. Maaradji, Eds., in Lecture Notes in Computer Science, Vol. 12632, Cham: Springer International Publishing, 2021, pp. 85-96, doi: 10.1007/978-3-030-76352-7_13

[7] M. Waseem *et al.*, "Understanding the Issues, Their Causes and Solutions in Microservices Systems: An Empirical Study," July 11, 2023, *arXiv*: arXiv:2302.01894. doi: 10.48550/arXiv.2302.01894

[8] S. Zhang *et al.*, "Failure Diagnosis in Microservice Systems: A Comprehensive Survey and Analysis," Jan. 14, 2025, *arXiv*: arXiv:2407.01710, doi: 10.48550/arXiv.2407.01710

[9] G. Yu *et al.*, "MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments," in *Proceedings of the Web Conference 2021*, Ljubljana Slovenia: ACM, Apr. 2021, pp. 3087-3098, doi: 10.1145/3442381.3449905

[10] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root Cause Localization of Performance Issues in Microservices," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, Budapest, Hungary: IEEE, Apr. 2020, pp. 1-9, doi: 10.1109/NOMS47738.2020.9110353

[11] D. Liu *et al.*, "MicroHECL: High-Efficient Root Cause Localization in Large-Scale Microservice Systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Madrid, ES: IEEE, May 2021, pp. 338-347, doi: 10.1109/ICSE-SEIP52600.2021.00043.

[12] C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu, "Eadro: An End-to-End Troubleshooting Framework for Microservices on Multi-source Data," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, Melbourne, Australia: IEEE, May 2023, pp. 1750-1762, doi: 10.1109/ICSE48619.2023.00150

[13] P. Kacsuk, R. Lovas, and J. Kovács, "Systematic Debugging of Parallel Programs in DIWIDE Based on Collective Breakpoints and Macrosteps1," in *Euro-Par'99 Parallel Processing*, vol. 1685, Springer Berlin Heidelberg, 1999, pp. 90-97, Accessed: Dec. 01, 2020 [Online] Available: https://doi.org/10.1007/3-540-48311-X_8

[14] R. Lovas and B. Vécsei, "Integration of Formal Verification and Debugging Methods in P-GRADE Environment," in *Distributed and Parallel Systems*, Vol. 777, Z. Juhász, P. Kacsuk, and D. Kranzlmüller, Eds., in The International Series in Engineering and Computer Science, Vol. 777, Boston: Kluwer Academic Publishers, 2005, pp. 83-92, doi: 10.1007/0-387-23096-3_10

[15] "Istio," Istio. Accessed: July 31, 2025 [Online] Available: https://istio.io/

[16] D. Lehmann and M. O. Rabin, “On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem,” in *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’81*, Williamsburg, Virginia: ACM Press, 1981, pp. 133-138, doi: 10.1145/567532.567547

[17] M. Héder *et al.*, “The Past, Present and Future of the ELKH Cloud,” *Inf. Társad.*, Vol. 22, No. 2, p. 128, Aug. 2022, doi: 10.22503/inftars.XXII.2022.2.8