Investigating Communication Overheads In Microservice Applications

Alberto Escobar Mingo
Supervised by: Reto Achermann and Michael Giardino
December 19th 2025

Abstract

This study demonstrates the communication overheads inherent in microservice architectures by systematically comparing Transmission Control Protocol (TCP) and Unix Domain Sockets (UDS) across various deployment configurations, including single-host, distributed bare-metal, and virtualized environments. We utilized two distinct microservice applications, the Social Network application from DeathStarBench and a independently developed Calculator application, to characterize performance across different levels of complexity.

Our results consistently show that TCP incurs significant latency overhead compared to UDS for intra-host communication, an effect that is dramatically amplified in simpler, low-latency microservices where the communication cost dominates the total latency budget. Furthermore, distributed deployments introduce massive additional overheads due to network stack traversal, with virtualization increasing this effect and consistently exhibiting the highest tail latencies. Finally, we demonstrate that a hybrid communication model, utilizing UDS for intra-host requests and TCP for inter-host requests, can reduce P99 tail latencies at high loads by approximately 8.02% compared to all-TCP distributed setups. This local optimization improves latency but does not significantly increase the overall saturation point, which remains constrained by the physical network link.

1. Introduction

Microservice architecture is a dominant paradigm in modern software systems. All cloud platforms, enterprise software platforms, and consumer software platforms utilize microservices as a way to divide up the functionality of their products into segregated microservices that talk to each other. When a user uses one of these platforms, any action will flow through a network of microservices to execute the user's action [1]. Since several microservices may be involved in a user's action its important that latency is minimized at each service to minimize the end to end latency a user may experience. More importantly it is important to not just focus on average latency but the latency experienced by 99th percentile of users.

While average latency provides a general performance indicator, the 99th percentile (P99) latency is often more critical in practice. P99 latency represents the worst experience that 1 in 100 users will encounter, and in high-traffic systems, this translates to millions of degraded user

interactions daily. Additionally, tail latencies are frequently amplified in microservice architectures due to request fanout, where a single user request triggers multiple downstream service calls, and the slowest response determines the overall latency.

Understanding communication overheads is critical for practitioners making deployment decisions in production environments. These overheads can account for a significant portion of end-to-end latency, and choosing the wrong deployment configuration can lead to unnecessary infrastructure costs or degraded user experience. By characterizing these overheads across different deployment scenarios, system architects can make informed tradeoffs between performance, resource utilization, and operational complexity

As mentioned earlier, user actions on a platform will flow through various microservices which means that each time a microservice receives or sends a request, the request will flow through a network stack inducing a latency due to communication overheads.These communication overheads directly contribute to end-to-end latency and can significantly degrade performance under high load.

This paper investigates how various deployment configurations impact communication overheads and latency in microservice applications. Deployment configurations that will be observed include microservice applications deployed across 2 hosts, across 2 virtual machines (each on a separate physical host), on 1 host using Transmission Control Protocol (TCP) for inter-service communication, and on 1 host using Unix Domain Sockets (UDS) for inter-service communication.

While prior work has examined microservice performance characteristics, this paper provides a systematic comparison of communication overheads across multiple deployment configurations using diverse benchmark applications. By measuring latency at the user end, service end-to-end, and per-service level under controlled conditions, we isolate the impact of different communication mechanisms and network topologies on p99 latency.

The microservice applications used in this paper will be the Social Network application from DeathStarBench, and a Calculator application. Deployment of applications was done only using Docker. Latency measurements were collected by analyzing trace durations using Jaeger distributed tracing

In Section 2 we provide some background information on microservices and related work. In Section 3 we describe equipment and methodology used. In Section 4 we present our results. In Section 5 we provide our discussion about the results, and section 6 we discuss future works. Finally in Section 7 we conclude our work.

2. Background

2.1 Microservices

Microservice architecture is a distributed system paradigm widely adopted in modern commercial software applications. It is characterized by decomposing an application into a collection of independently deployable, loosely coupled services that communicate with each other. This approach contrasts sharply with the traditional monolithic architecture, where all functional components are bundled into a single, tightly-integrated unit. Excellent real-world examples of platforms utilizing this paradigm include Netflix and Facebook.

In a typical microservice environment, individual services are placed within isolated environments, most commonly Docker containers, and managed by a container orchestrator (e.g., Kubernetes or Docker) for deployment, scaling, and operational efficiency. Each container is configured to communicate with necessary downstream services, forming a complex network. A user request typically begins at a frontend service and then fans out, triggering a cascade of inter-service calls to fulfill the final action.

Services primarily use network protocols like Transmission Control Protocol (TCP) for inter-service communication. TCP allows services to communicate seamlessly whether they reside on the same physical host or across different hosts in a network. This loose coupling offers several significant benefits, including [2]:

- Separation of responsibilities where there is clear division of features into distinct services.
- Independent development and management of services so teams can modify, deploy, and scale individual services without impacting the system as a whole.
- Technology diversity by implemenint services using different programming languages and databases tailored to their specific needs.

However, a well-known drawback to microservice architectures is the increased latency due to communication overheads [2]. Since a single user action now passes through multiple services, each interaction involves the full network stack to send and receive requests. Monolithic applications mitigate this problem by eliminating the network stack for internal calls, often utilizing much faster Inter-Process Communication mechanisms.

2.2 Related Works

Detti et al. introduced μBench, an open-source tool designed to generate microservice applications for benchmarking cloud and edge computing platforms [3]. The tool provides researchers with a factory to create diverse service mesh topologies and simulate specific microservice behaviors by generating loads and targeting specific computational resources. In their study, the authors utilized μBench to compare the trade-offs between monolithic and microservice architectures, specifically analyzing how architectural choices directly impact

application performance. By evaluating these generated applications, their work signals the significant role that topology and inter-service communication play in determining the overall latency and efficiency of distributed systems.

Kurpad et al. conducted an analysis to quantify the performance overheads introduced by service meshes within a Kubernetes environment [4]. Using the Istio service mesh and the DeathStarBench suite, the authors utilized tools such as wrk2 for latency sensitivity and Linux-perf to examine low-level system impacts. Their research provides a granular view of how the dedicated infrastructure layer of a service mesh affects CPU cycles, system instructions, page faults, and cache misses. This work is relevant as it demonstrates the significant overheads incurred when deploying microservices onto a host and how those low-level costs translate into degraded performance for the microservice application.

Yashu et al [5] explores a promising solution to the latency challenges of microservices in the work. The researchers introduce MPKLink, a system designed to bridge the performance gap between monolithic IPC and microservice networking. By leveraging Intel Memory Protection Keys (MPK), MPKLink enables co-located microservices to communicate via high-speed shared memory rather than traditional network protocols like REST or gRPC. While shared memory is inherently faster, it historically posed security risks; however, MPKLink uses hardware-level access control to ensure that services remain isolated and secure. This approach effectively eliminates the overhead of the network stack for intra-container communication.

2.3 Microservice Benchmarks

To ensure the generalizability of our findings, this study employs two distinct microservice applications that represent different scales of complexity and service density.

The most prominent suite of microservice benchmarks is DeathStarBench [6], developed by researchers at Cornell University. This suite includes several microservice applications designed to simulate real-world cloud workloads. For this paper, we utilized the Social Network application, which serves as a functional replica of a platform like Facebook.

The Social Network architecture is highly complex, comprising 37 unique services. It allows for a wide range of user interactions, including account creation, status updates with multimedia content, and social graph interactions such as following users or liking posts. To maintain state and performance, it leverages several off-the-shelf services, including NGINX, Redis, Memcached, and MongoDB.

In contrast to the high complexity of DeathStarBench, we also utilized a more streamlined application called Calculator. Inspired by the architectural patterns of the TeaStore benchmark [7], Calculator is a lightweight microservice application written in Go.

The application consists of 7 services, including a Memcached container for caching results. Users interact with a frontend service to perform basic arithmetic operations (addition,

subtraction, multiplication, and division) on two operands. A key design feature of this application is the abstraction of the communication layer, which allows for seamless switching between TCP and UDS via configuration on the docker file used for deployment [8]. Figure 1 depicts the architecture of the Calculator application.

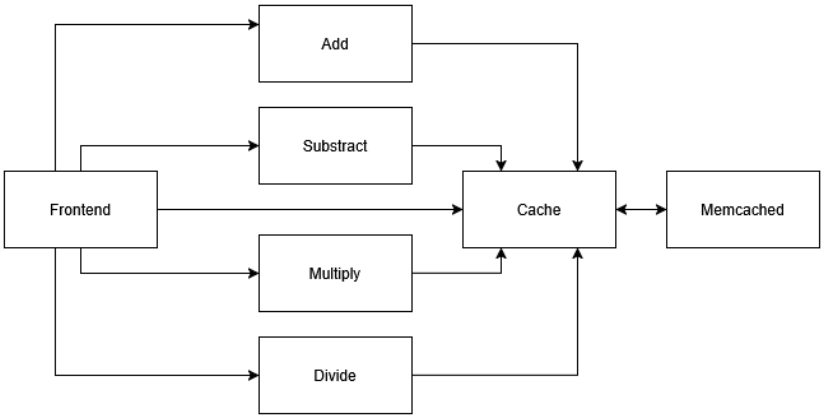

Figure 1. Architecture Diagram for Calculator Application.

When deployed using UDS, the services communicate by sharing a Docker volume that hosts the necessary socket files. This setup allows us to measure the performance benefits of bypassing the network stack while maintaining container isolation.

3. Methodology

3.1 Experimental Hardware and Network Baseline

All experiments were conducted on a dedicated cluster of three identical physical machines, leap07, leap08, and leap09, to ensure a consistent environment. The specifications for each machine are shown in table 1.

Table 1. Technical Specifications of the Physical Computers.

Component	**Specification**
CPU	Intel(R) Xeon(R) Silver 4309Y @ 2.80GHz (32 Cores)
RAM	64 GiB DDR4
NIC	NetXtreme BCM5720 Gigabit Ethernet PCIe
Operating System	Ubuntu 24.04.3 LTS

A network baseline was established by measuring communication characteristics between the host machines using Matt's Traceroute (mtr) and Iperf (iperf3). The baseline measurements confirmed low-latency and high-bandwidth connectivity and can be seen in table 2.

Table 2. Network Performance Measurements between Physical Computers.

Average Network Latency	0.2 ms
Number of Hops	1 hop
Average Bandwidth	931 Mbps

Virtuals machines were also used in the experimental work for this paper (further explained in section 3.3). Table 3 describes the specifications used to construct the virtual machines using QEMU and table 4 describes the baseline network measurements between virtuals machines. The hostnames given to the virtual machines are identical to the hostnames of the computer the virtual machine runs on to simplify the deployment of applications.

Table 3. Technical Specifications of the Virtual Machines.

Component	**Amount**
CPU	16 vCPUs
RAM	32 GB
Operating System	Ubuntu 25.10

Table 4. Network Performance Measurements between Virtual Machines.

Average Network Latency	0.4 ms
Number of Hops	1 hop
Average Bandwidth	931 Mbps

3.2 Benchmark Applications and Software Stack

Two microservice applications were chosen to represent different architectural complexity and communication patterns:

1. DeathStarBench Social Network: A complex, real-world suite with high service fan-out and intricate communication paths, representing a typical large-scale social media platform [6].
2. Calculator Application: A simple, application in developed Go, used to isolate and measure the raw communication overhead with minimal application-level logic interference.

The deployment and orchestration for all benchmark applications were handled exclusively using Docker[9]. This standardized the containerization across all test scenarios. Load generation was performed using wrk2, depending on the supplied code from the specific benchmark application. The justification for using wrk2 is its wide adoption in DeathStarBench and its use in research by Kurpad et al [4].

For the 1-host UDS deployment, we modified the Social Network application to utilize UDS instead of standard TCP-based communication. This modification allowed us to observe the impact of IPC optimizations on massive-scale architectures. However, due to specific configuration limitations, not all connections were replaced. notably, connections between the NGINX web server and backend services retained their standard TCP configuration, resulting in a hybrid communication model for this specific deployment.

3.3 Deployment Configurations

To systematically quantify communication overheads, experiments were performed across four distinct deployment configurations, utilizing machines leap07 and leap09 for the application services.

1. 1 Host TCP: All microservices deployed on a single host (leap07), with service mesh communication over TCP.
2. 1 Host UDS: All microservices deployed on a single host (leap07), with service mesh communication over UDS.
3. 2 Hosts: Services distributed across two separate physical machines (leap07 and leap09).
4. Virtual Machines: Services distributed across two Virtual Machines (VMs), with each VM residing on a separate physical host (leap07 and leap09). This measures the combined overhead of virtualization and physical network traversal.

For deployments conducted on two hosts (2 Hosts and Virtual Machines) microservices responsible for persistence or caching (e.g., databases, caches) were hosted one machine, while the remaining application services ran on the other. This was done to ensure most recorded traces required service requests to traverse the physical network and quantify the overhead of physical network traversal. Figure 2 shows the four different deployments used in this paper.

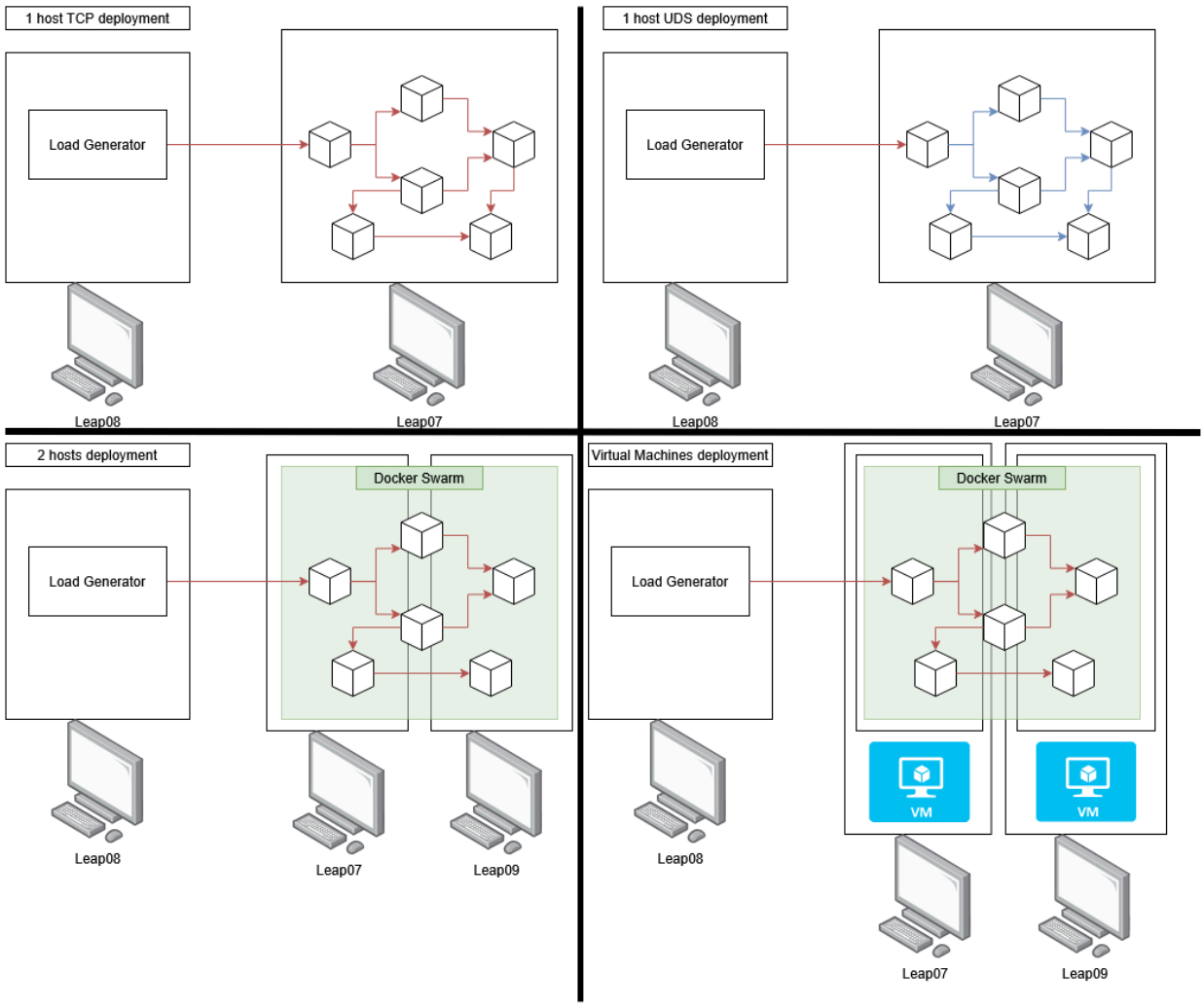

Figure 2. Four configurations for deploying a microservices application. Deployments are categorized by their hosting environment and the primary communication protocol used: Top left 1 Host TCP Deployment and top right 1 Host UDS Deployment compare TCP vs. UDS communication on a single host. Bottom left 2 Hosts Deployment and bottom right Virtual Machines Deployment showcase distributed deployment of containers, contrasting bare-metal vs. Virtual Machine hosting for the microservices application. The Load Generator resides on Leap08 across all scenarios. Red arrows indicate TCP connections, and blue arrows indicate UDS connections.

3.4 Data Collection and Analysis

Load generation was performed using the third dedicated physical machine, leap08, to ensure load generation computation performance did not interfere with application computation performance. Each experimental run was executed for a period of 10 minutes at a constant applied load measured in requests per second (RPS).

Latency data was collected using Jaeger distributed tracing, which was integrated into all applications. Jaeger was selected because the DeathStarBench applications are pre-configured with it, and it has been utilized in microservices esearch by Anatoly Krasnovsky [6][10]. Following the 10-minute load period, a sample of 5,000 most recent complete traces originating from the frontend service was collected from the Jaeger backend for detailed analysis. All latencies are measured in milliseconds. The latency collected from the Jaeger backend is referred to as “service processing latency” since it only observes the latency it takes from a request that arrives at the entry point of the application until its exit. Latency recorded from the load generator is also presented in the results as it observes the latency perceived from clients and is referred to as “end to end latency”.

The collected traces were processed using custom Python scripts to perform the following analyses, focusing specifically on tail latencies:

- Trace Latency Histograms: Generating a histogram of all service procesing trace latencies at a specific, representative applied load to visualize the distribution and spread of response times. Mean, P95, and P99 latencies are denoted on the histograms.
- Latency vs. Applied Load: Plotting P99, P95, and Mean service processing latency against increasing applied load (RPS) for both the service processing latency (from frontend to final response) and the end to end latency (as observed by the load generator). System saturation is set when P99 latencies are at 1000 ms.

4. Results

4.1 Social Network Application Performance Analysis

4.1.1 Tail Latency Comparison Across Deployment Configurations

The Social Network application was tested at low load of 100 RPS and high load of 2500 RPS in all four deployments. Table 5 summarizes the mean and tail service processing latencies at 100 RPS with figure 3 showing the service processing latency histograms of the traces. Table 6 summarizes the the mean and tail service processing latencies at 2500 RPS with figure 2 showing the service processing latency histograms of the traces.

Table 5. Social Network Service Processing Latency Summary at 100 RPS.

Deploment	**Mean Latency (ms)**	**P95 Latency (ms)**	**P99 Latency (ms)**	**% Change of P99 Latency**
1 host UDS	4.51	13.49	18.78	0.00
1 host TCP	4.75	13.65	19.20	2.24
2 hosts	7.76	19.71	27.18	44.73
virtual machines	22.5	52.90	66.47	253.94

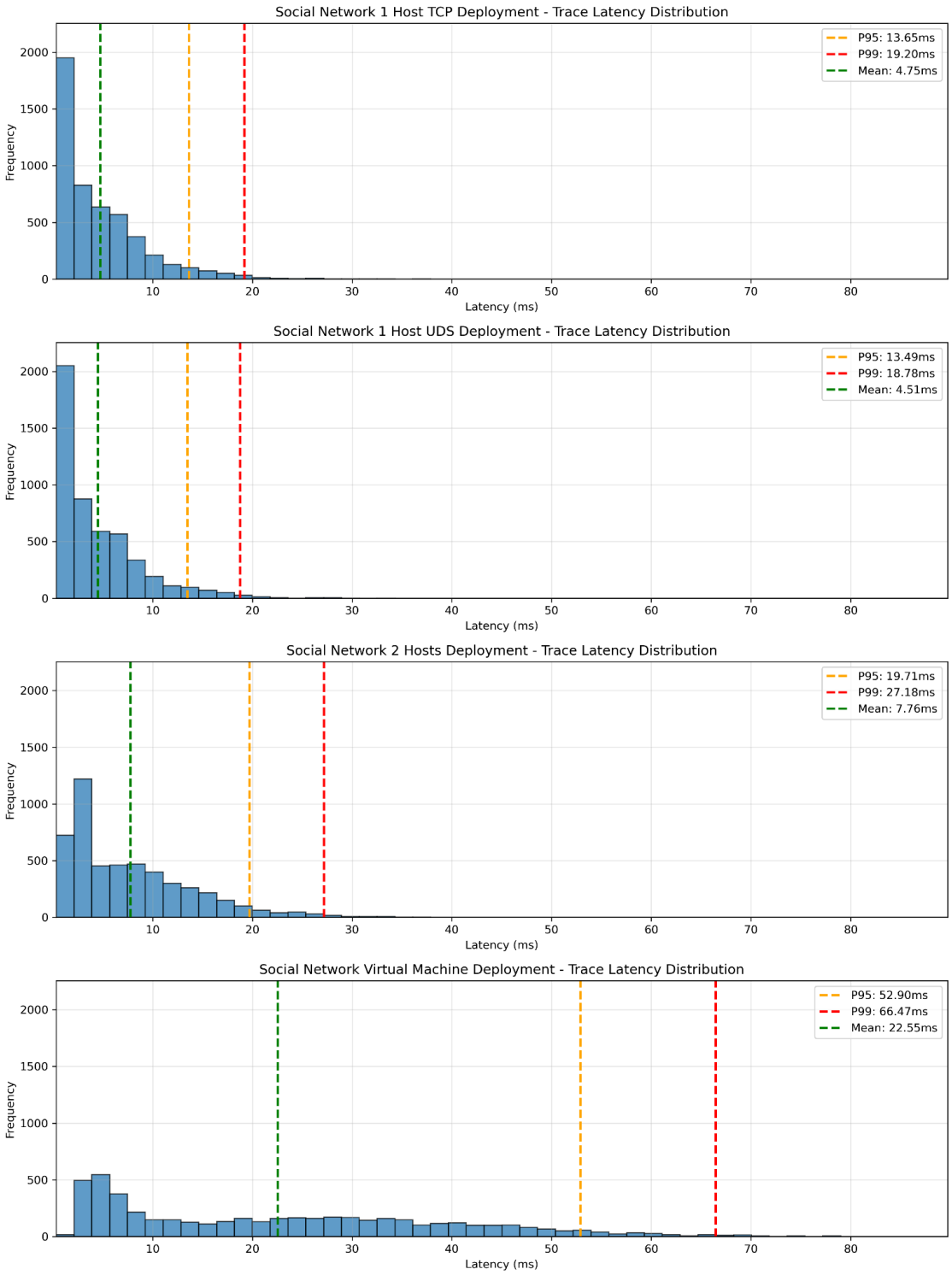

Figure 3. Social Network Service Processing Latency Histrograms at 100 RPS.

At 100 RPS, the applied load is low enough that the effects of queueing in the communication stack are minimal. When comparing the intra-host deployments, using the 1 host UDS deployment as the performance baseline, the 1 host TCP deployment exhibits a small P99 tail

latency increase of 2.24%. When inter-service requests are forced to traverse a physical network in the 2 host deployment, the P99 tail latency increases significantly by 44.73%. The highest latency is recorded in the Virtual Machines deployment, where the combined effect of virtualization and physical network traversal results in a massive 253.94% increase in P99 tail latency.

Table 6. Social Network Service Processing Latency Summary at 2500 RPS.

Deplоment	**Mean Latency (ms)**	**P95 Latency (ms)**	**P99 Latency (ms)**	**% Change of P99 Latency**
1 host UDS	124.40	280.92	388.66	0.00
1 host TCP	168.98	352.29	452.68	16.47
2 hosts	275.85	1188.78	1724.40	343.68
virtual machines	365.82	916.70	1575.28	305.31

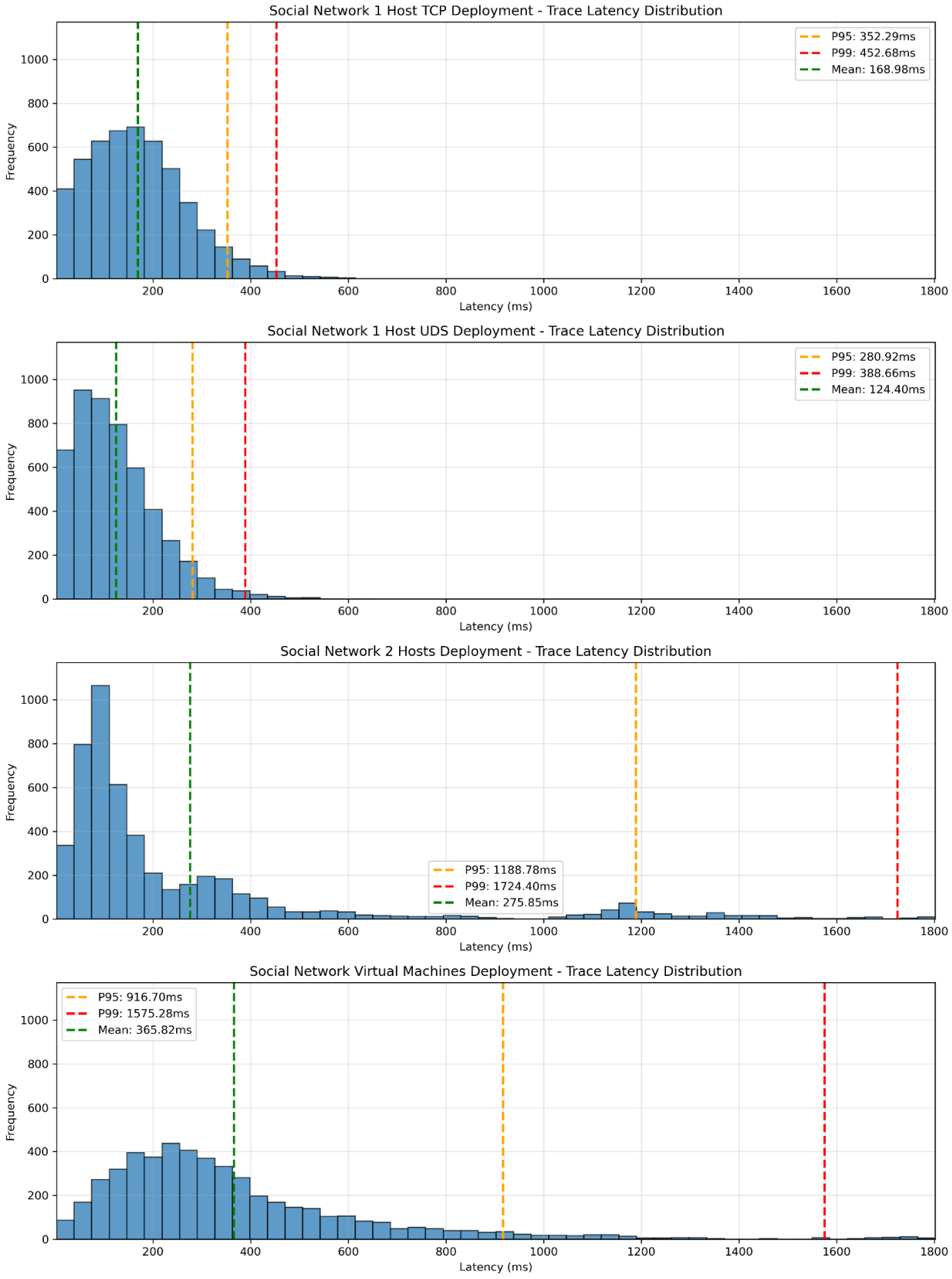

Figure 4. Social Network Service Processing Latency Histograms at 2500 RPS.

At 2500 RPS and using the 1 host UDS deployment as the performance baseline, the latencies of the 1 host TCP deployment is amplified, resulting in a 16.47% increase in P99 tail latency. Introducing inter-host communication causes a spike in tail latency. The 2 hosts deployment shows a substantial P99 tail latency increase of 343.68%. Finally, the Virtual Machines deployment, while still exhibiting a substantial P99 increase of 305.31%, surprisingly shows a lower P99 latency of 1575.28 ms than the 2 host deployment of 1724.40 ms. However, the mean latency for the Virtual Machines deployment is 365.82 ms, which remains higher than the 2 hosts deployment mean latency of 275.85 ms.

4.1.2 Impact of Applied Load on Tail Latency

Tail latencies were recorded across a range of requested loads from 100 to 4000 RPS to characterize the saturation point of each deployment. The measurements include service processing latency (time spent within the application services, measured by Jaeger traces) and end-to-end latency (time observed by the load generator for the full request-response cycle). Figures 5, 6, 7, 8 display the Service Processing P99 and End-to-End P99 latencies against the applied load for the four deployment configurations.

It is important to note that for requested loads at or above 3000 RPS, the load generator was unable to achieve the target throughput due to back-pressure mechanisms within the system, resulting in throughput throttling. For visualization purposes, graphs were capped at 2000 ms to effectively display the performance before the onset of instability, as some latencies spiked to ~50 seconds. Furthermore, in cases where the End-to-End P99 latency appears lower than the Service Processing P99 latency, this is attributed to statistical sampling differences, as the service processing data relies on a sample of 5,000 traces from the Jaeger backend, while the end-to-end data accounts for all requests recorded by the load generator.

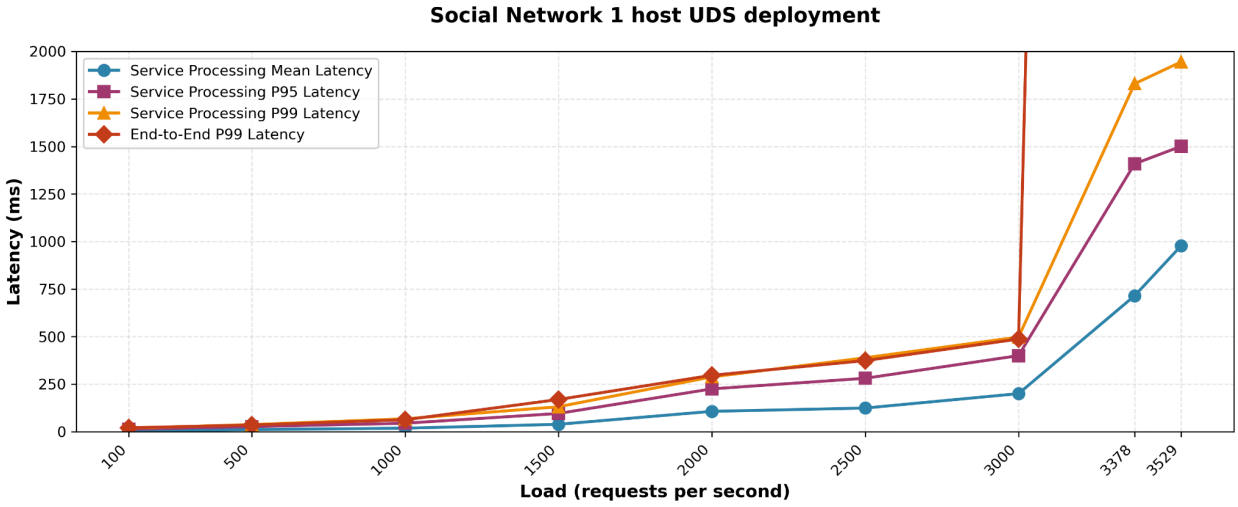

Figure 5. Social Network 1 Host UDS Deplomeent Latency vs. Load Graph.

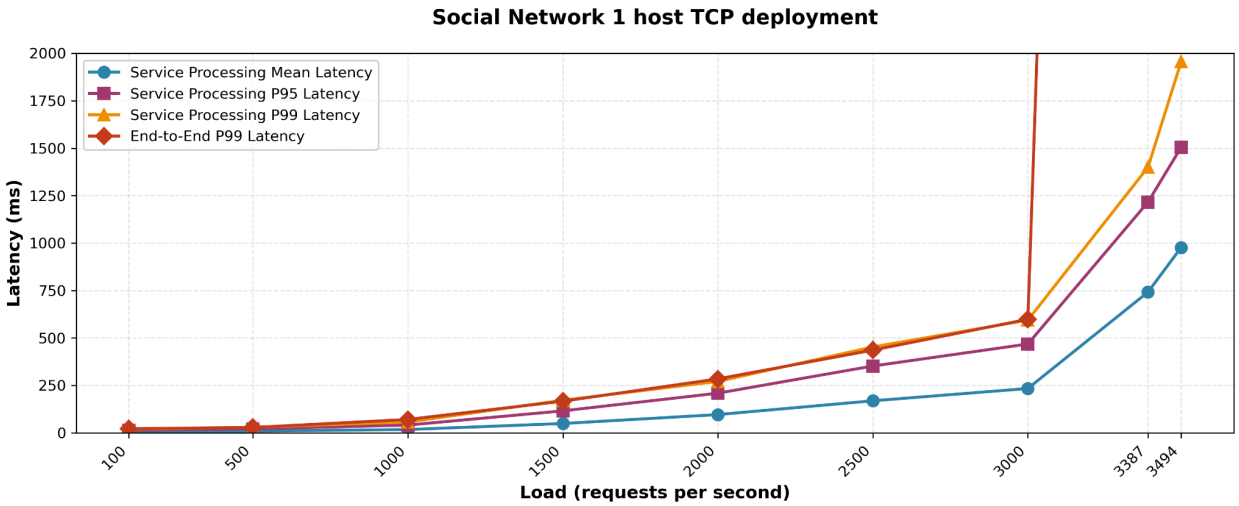

Figure 6. Social Network 1 Host TCP Deployment Latency vs. Load Graph.

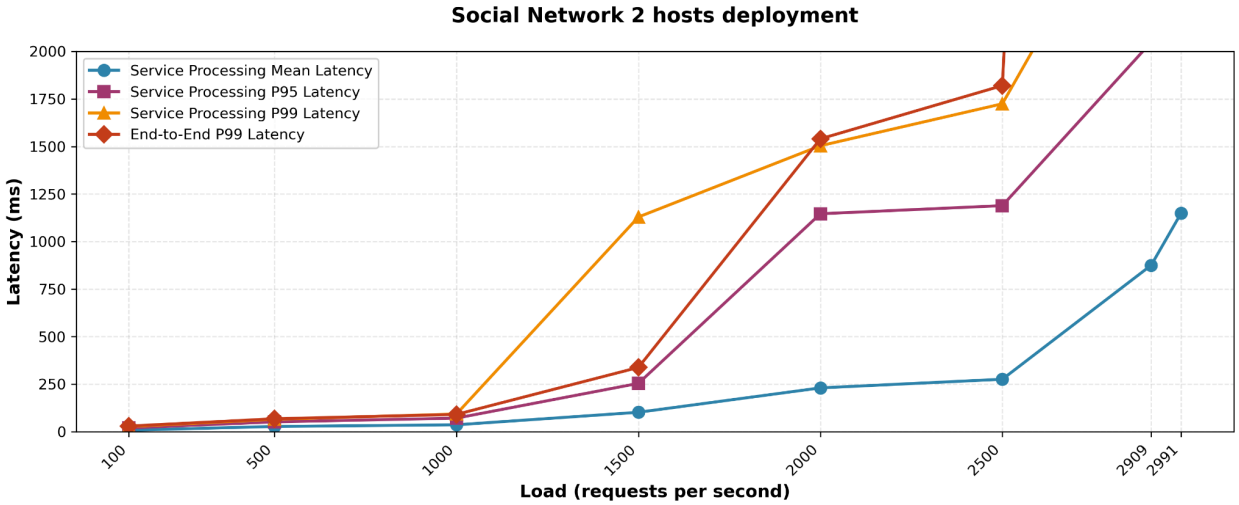

Figure 7. Social Network 2 Hosts Deployment Latency vs. Load Graph.

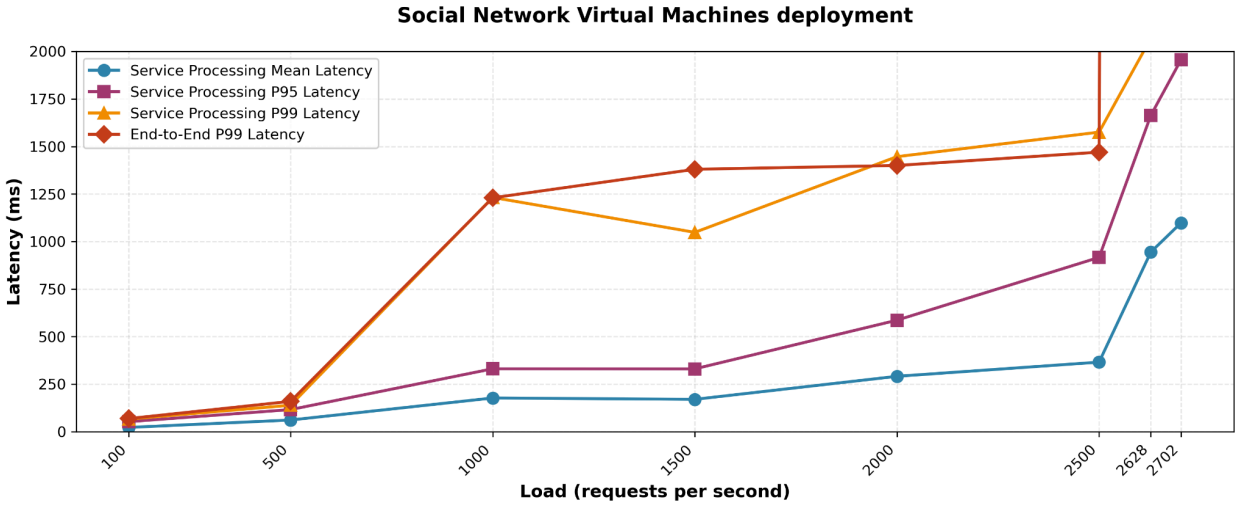

Figure 8. Social Network Virtual Machines Deployment Latency vs. Load Graph.

The single host deployments (1 host UDS and 1 host TCP) demonstrate the highest stability and capacity. Latencies in both setups remain well below 500 ms up to 3000 RPS. A dramatic increase in P99 latency occurs after 3000 RPS. Crucially, the 1 host UDS deployment consistently maintains lower P99 latency across all loads before this overload point compared to the TCP counterpart. Throughput throttling is observed when requesting 3500 RPS, in both deployments. The capacity of the Two Hosts deployment is significantly reduced. P99 latencies begin to spike rapidly after the 1000 RPS mark. Throughput throttling is observed when requesting 3000 RPS, which is a slightly lower effective load limit than the 1 host deployments achieved. The Virtual Machines deployment shows the poorest scaling behavior. P99 latency increases dramatically after only 500 RPS. The throughput throttling begins when requesting 3000 RPS.

4.2 Calculator Application Performance Analysis

4.2.1 Tail Latency Comparison Across Deployment Configurations

The Calculator microservice application, which represents a simple was benchmarked across the four deployment configurations at a low applied load of 100 RPS and a high applied load of 1000 RPS. Table 6 summarizes the tail latencies, with Figure 9 showing the corresponding histograms at 100 RPS. Table 7 summarizes the tail latencies, with Figure 10 showing the corresponding histograms at 1000 RPS.

Table 6. Calculator Service Processing Latency Summary at 100 RPS.

Deploment	Mean Latency (ms)	P95 Latency (ms)	P99 Latency (ms)	% Change of P99 Latency
1 host UDS	0.6	1.35	1.91	0.00
1 host TCP	1.93	3.83	5.86	206.81
2 host	8.06	20.84	24.21	1167.54
virtual machines	64.52	114.08	134.13	6922.51

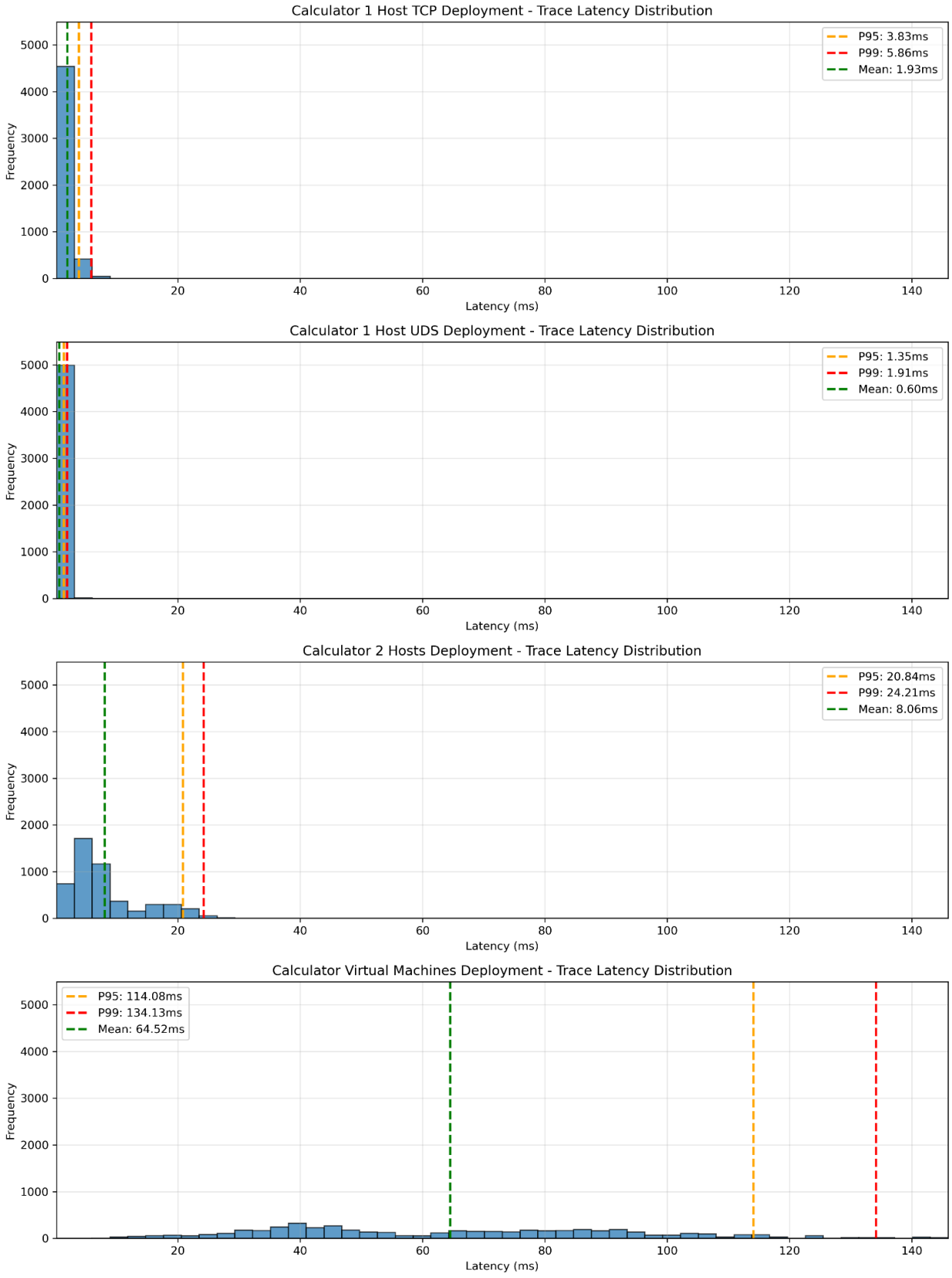

Figure 9. Calculator Service Processing Latency Histrograms at 100 RPS.

At 100 RPS when, comparing the intra-host deployments, using the 1 host UDS deployment as the performance baseline, the 1 host tcp deployment exhibits a P99 tail latency increase of 206.81%. Introducing a physical network traversal within the microservice application greatly increases latency. The 2 host deployment sees the P99 tail latency jump by 1167.54%. The highest latency is again recorded in the virtual machines deployment with a P99 tail latency increase of 6922.51%.

Table 7. Calculator Service Processing Latency Summary at 1000 RPS.

Deploment	Mean Latency (ms)	P95 Latency (ms)	P99 Latency (ms)	% Change P99 Latency
1 host UDS	0.97	3.89	7.4	0.00
1 host TCP	7.77	27.43	33.41	351.49
2 host	24.87	189.51	257.73	3382.84
virtual machines	1905.05	2783.48	2993.6	40354.05

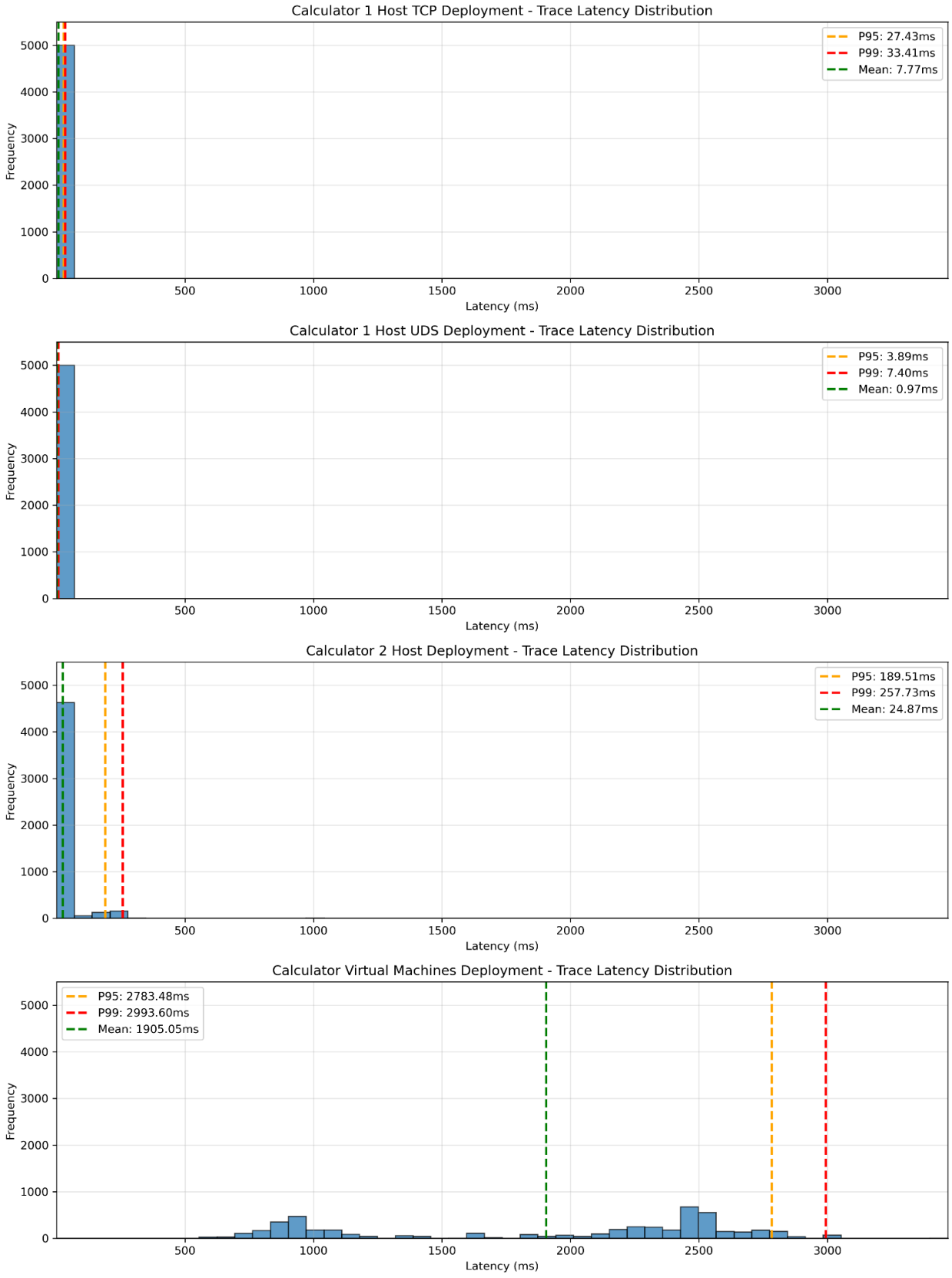

Figure 10. Calculator Service Processing Latency Histograms at 1000 RPS.

At 1000 RPS, using the 1 host UDS configuration as the baseline, the overhead of TCP in the 1 host TCP deployment is substantially amplified under high load, resulting in a 351.49% increase in P99 tail latency. The inter-host deployments show an extreme increase in tail latency. The 2 host deployment experiences a P99 increase of 3382.84%, with P99 latency increasing to 257.73 ms. In the Virtual Machines deployment, the P99 tail latency soars to 2993.64 ms, representing an increase of 40354.05%.

4.2.2 Impact of Applied Load on Tail Latency

Tail latencies for the Calculator application were recorded across a range of requested loads from 100 to 2250 RPS to characterize the saturation point of each deployment. As before, the measurements include service processing latency (traced time within the application services) and End-to-End Latency (full request-response time observed by the load generator). Figures 11 to 14 show tail latency growth curves for the four deployment configurations. For visualization purposes, graphs were capped at 5000 ms to effectively display the performance before the onset of instability, as some latencies spiked to ~60 seconds.

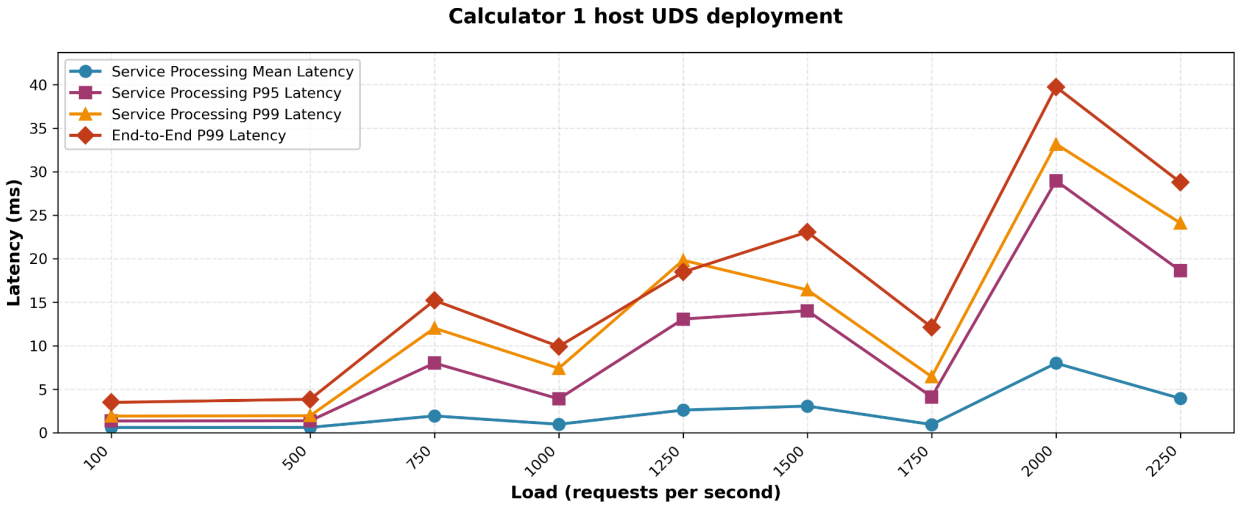

Figure 11. Calculator 1 Host UDS Deplomemt Latency vs. Load Graph.

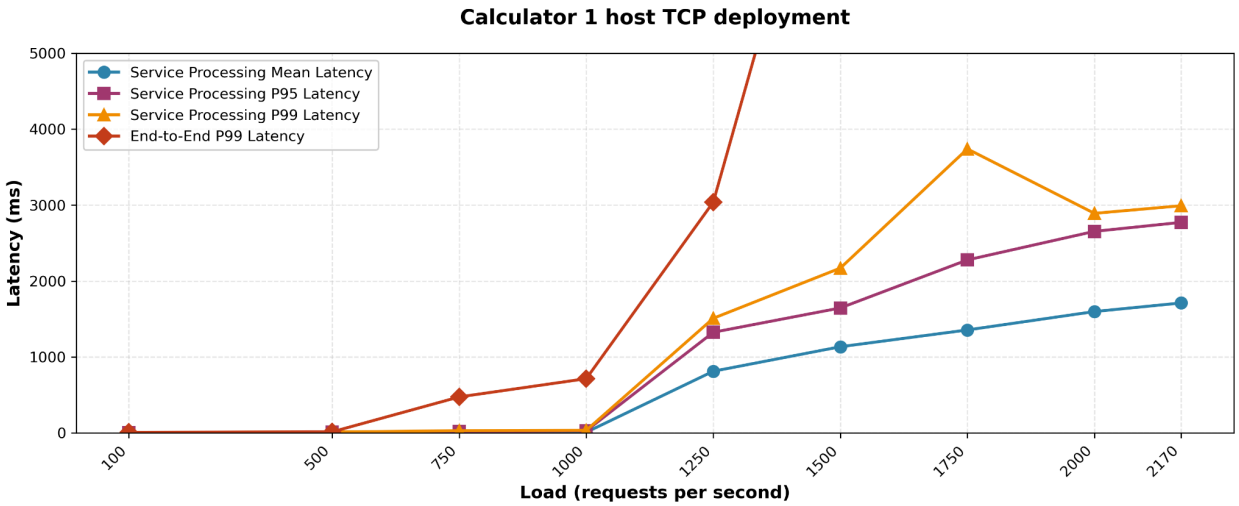

Figure 12. Calculator 1 Host TCP Deployment Latency vs. Load Graph.

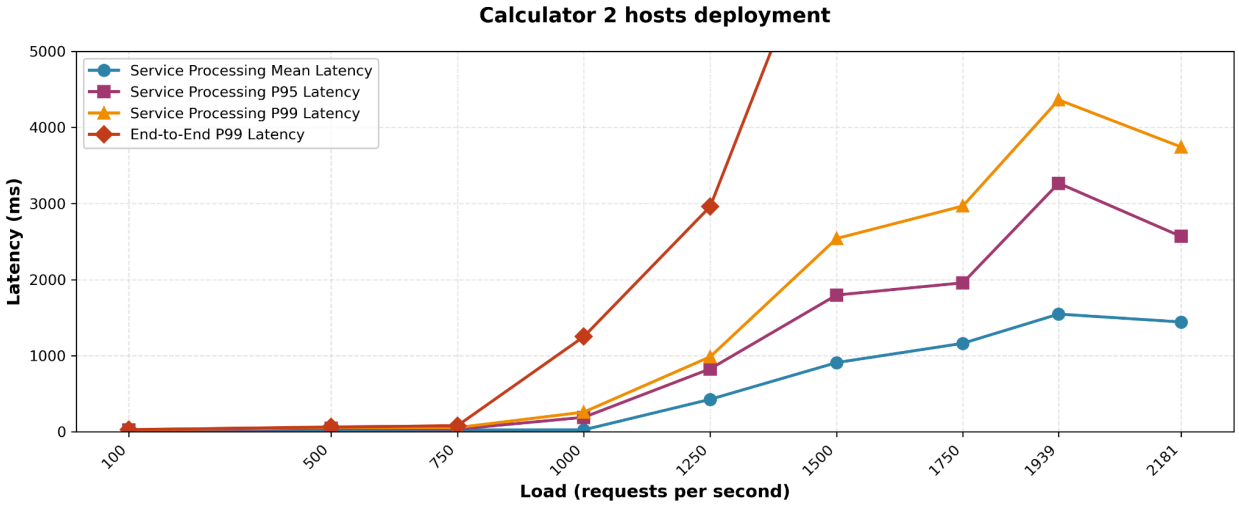

Figure 13. Calculator 2 Hosts Deployment Latency vs. Load Graph.

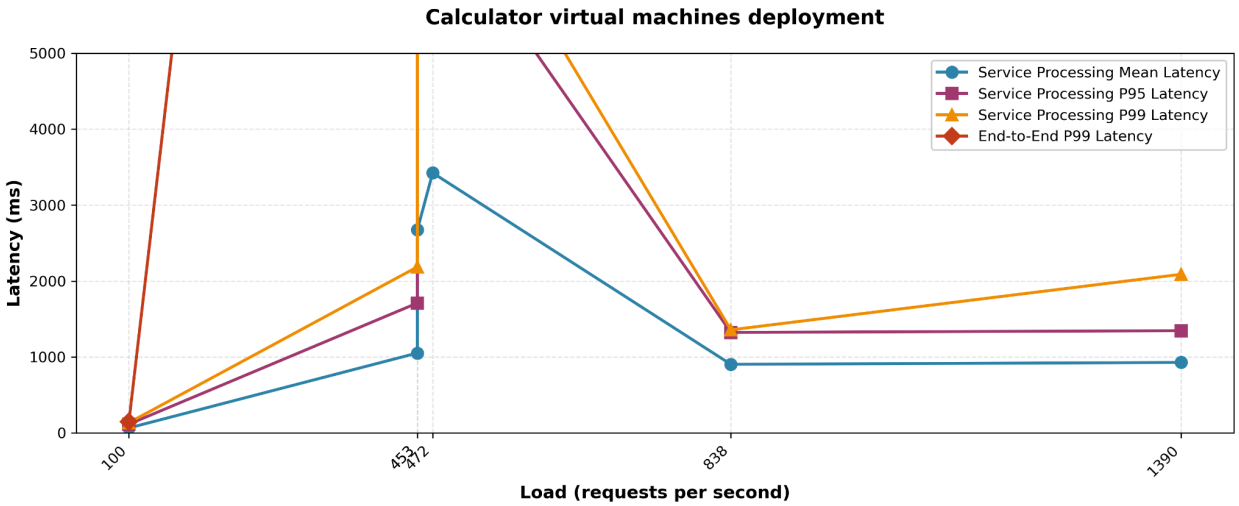

Figure 14. Calculator Virtual Machines Deplomment Latency vs. Load Graph.

The 1 host UDS deployment demonstrates exceptional stability and the lowest overall latency. Service Processing P99 latency remains below 20 ms across the entire tested range of 100–2250 RPS. End-to-End P99 latency remains below 40 ms. In the 1 host TCP deployment, performance degrades rapidly. The saturation threshold is clearly passed after 1000 RPS, where the End-to-End P99 latency spikes drastically from 713.73 ms to 3040 ms. Similarly, the Service Processing P99 latency exceeds 1000 ms at 1250 RPS.

Performance degradation is observed at even lower loads in the distributed setups. For the 2 host deployment, End-to-End P99 latency spikes sharply after 750 RPS (reaching 1250 ms at 1000 RPS), and Service Processing P99 latency begins to increase rapidly after 1000 RPS. The results for the Virtual Machines deployment are the most dramatic, highlighting the combined impact of virtualization and network traversal. Both Service Processing and End-to-End P99 latencies exceed 1000 ms starting after 1000 RPS. A notable discontinuity in the data is observed at 1500 RPS, where latencies improve. Given the extreme variability and high values recorded in this setup, this discontinuity could indicate moments of instability in the deployment or load generator during experimentation resulting in unreliable results.

4.3 Hybrid Deployment Performance Analysis

This final section investigates a Hybrid deployment configuration for the Calculator application where the application services are distributed across two physical hosts, but communication is optimized using a mix of protocols. As illustrated in Figure 15, services communicating within the same host utilize UDS, while services that communicate between the two separate hosts use TCP. This design aims to combine the performance benefits of UDS for intra-host calls while also using TCP for inter-host requests between services allowing for scalability.

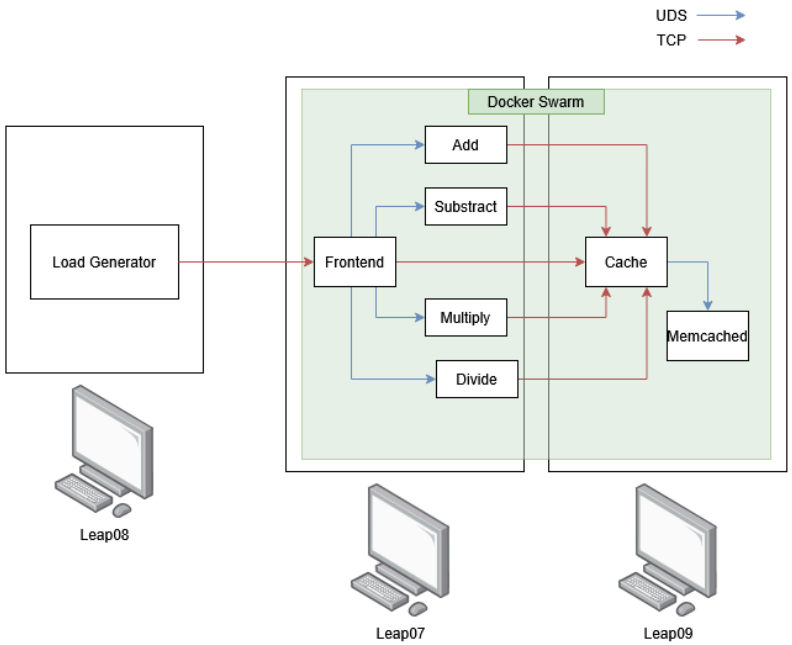

Figure 15. Visual Diagram describing the 2 host hybrid deployment of the calculator application.

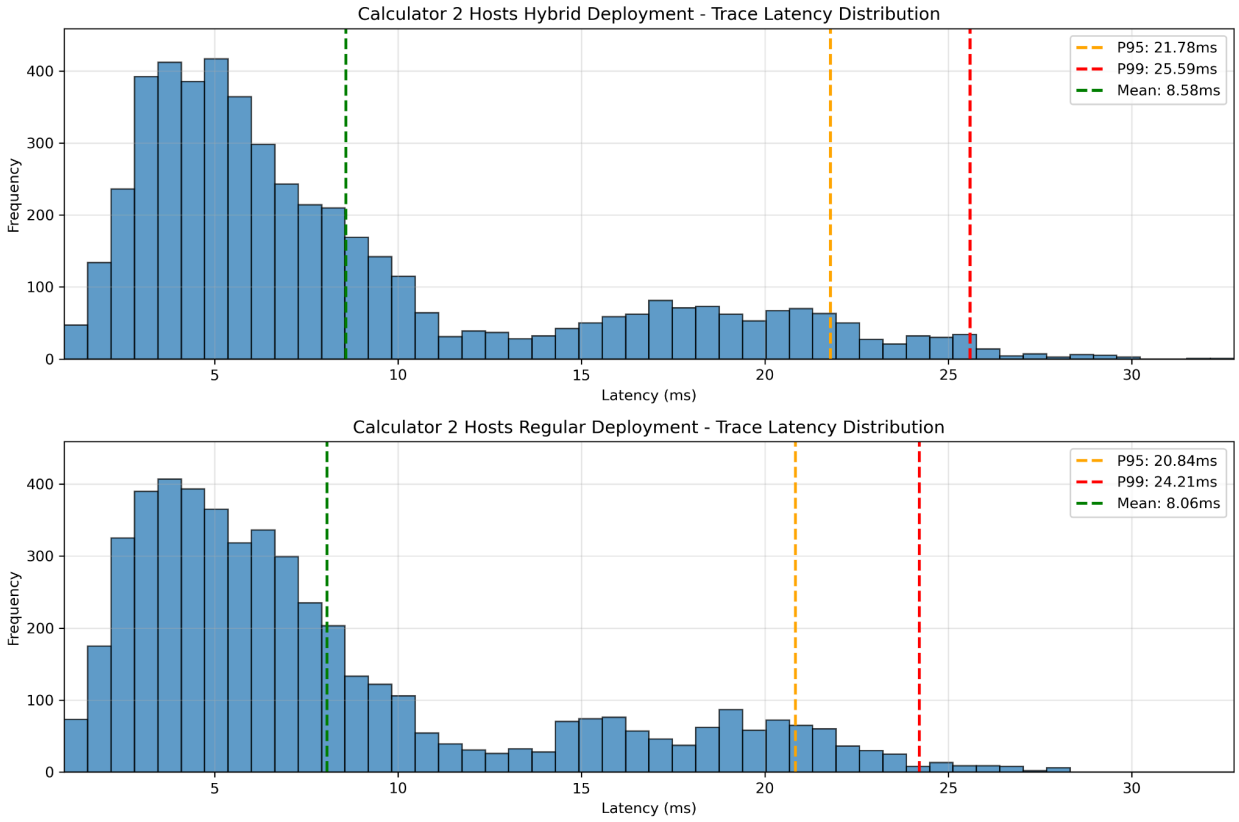

Figure 16. Service Processing Latency Histrograms of Calculator 2 Hosts Regular and Hybrid Deployment at 100 RPS.

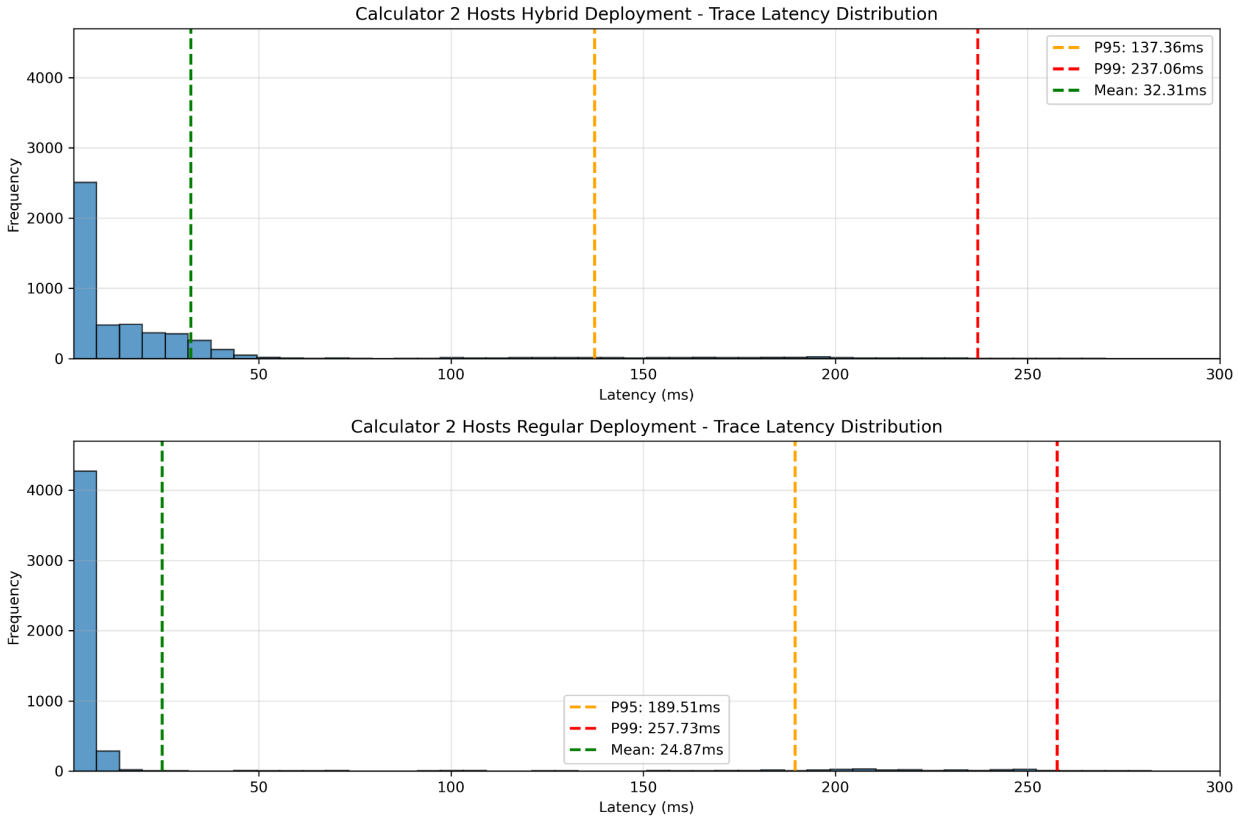

Figure 17. Service Processing Latency Histrograms of Calculator 2 Hosts Regular and Hybrid Deployment at 1000 RPS.

As shown in the trace histograms in figure 16, the performance difference between the 2 host hybrid deployment and the 2 Host regular deployment is minimal at 100 RPS. The Hybrid system exhibits slightly higher Mean, P95, and P99 latencies (less than 1 ms difference across all metrics). In figure 17, a noticeable performance gain is seen in the 2 host Hybrid deployment at 1000 RPS. The Hybrid system shows an 8.02% improvement in P99 latency compared to the regular 2 Host deployment. The P99 tail latency drops from 257.73 ms to 237.06 ms .

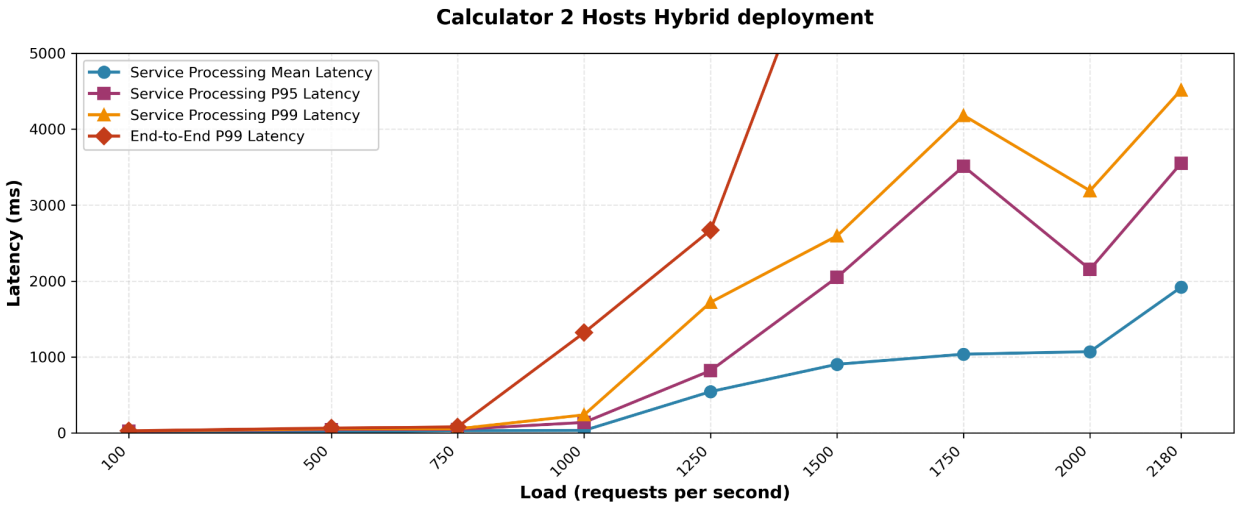

Figure 18. Calculator 2 Host Hybrid Deplomент Latency vs. Load Graph.]

When testing the Hybrid deployment across the full range of loads (100 RPS to 2180 RPS, Figure 18), the system's saturation profile is clear. Similar to the other distributed setups, the Hybrid application begins to get overwhelmed rapidly, with latency dramatically increasing after 750 RPS. The saturation point is reached at approximately 1250 RPS, where the End-to-End P99 Latency spikes above 2500 ms and service processing latency is above 1000 ms. Although the Hybrid approach offers a measurable improvement in P99 latency at high load compared to the regular 2 Host deployment, it does not significantly delay the onset of saturation.

5. Discussion

5.1 Protocol Overheads: UDS vs. TCP

The results consistently demonstrate the communication overhead of TCP compared to UDS for intra-host service mesh communication. In the single host configuration (1 host deployment) across both the Calculator and Social Network applications, UDS exhibits lower P99 service processing latency compared to TCP. This difference demonstrates the overheads introduced by using TCP connections. The magnitude of this overhead varies greatly by application and load. At low load, the overhead resulted in an increase in P99 service processing latency of 2.24% for the Social Network. However, for the simple Calculator application, the TCP overhead accounted for a massive 206.81% increase. This demonstrates that for services with very low intrinsic processing time, the fixed cost of the TCP stack consumes the majority of the latency budget. Under high load, the TCP overhead is amplified, with the P99 service processing latency difference soaring up to 16.47% in the Social Network application and 1351.49% for the Calculator application. At these high throughputs, the system's capacity is constrained, and the overhead from the TCP stack quickly translates into substantial queueing delay causing the accelerated degradation of tail latencies as services struggle to keep up with the processing demand.

The difference between the Social Network and Calculator performance change can be attributed to the complexity and inherent processing time of the services within each application. The Social Network is comprised of 36 unique services [6], requiring user requests to flow through multiple services and perform more complex operations. Conversely, the Calculator Application is simpler in design, comprised of 7 services, with user requests only traversing a handful of services before completion. The Social Network's higher baseline processing time effectively dampens the relative impact of the communication overhead: the TCP cost is a smaller fraction of the total latency. In contrast, the Calculator's minimal processing time makes its latency overwhelmingly sensitive to communication costs, resulting in the massive relative percentage increases observed when using TCP.

When the applications are deployed across a physical network (the 2 Host configuration), the communication overheads escalate dramatically. While the baseline network latency between the hosts is minimal at 0.2 ms, the observed P99 service processing latency increases range from 44.73% (Social Network, low load) to an extreme 3382.84% (Calculator, high load). For the Social Network application at high loads, the P99 service processing latency reached 1724.40 ms in the 2 Host deployment, starkly contrasting with 388.66 ms in the 1 host UDS deployment at 2500 RPS. This massive degradation cannot solely be attributed to propagation delay, but is due to the compounded overhead of network stack traversal and resulting congestion. Every inter-service request must traverse the full network stack twice (down the stack on the sender, and back up the stack on the receiver). This involves serialization, context switching, and the entire packet processing pipeline in the kernel. Under high load, the physical Network Interface Card and its associated queues become severe bottlenecks. Requests queue at these points, and the slowest request in the fan-out chain dictates the overall P99 latency, leading to the rapid saturation seen in the load vs. latency curves.

The analysis of latency across varying loads in Sections 4.1.2 and 4.2.2 reveals insights into the relationship between application complexity and communication overhead. In the Social Network application, both the UDS and TCP single-host deployments maintain stability until reaching a saturation point beyond 3000 RPS. In stark contrast, the Calculator application demonstrates a much higher sensitivity to the communication stack; while the single-host UDS deployment shows no signs of saturation within the tested range, the single-host TCP deployment reaches saturation shortly after only 1000 RPS. This discrepancy suggests that communication overheads have a less significant impact on the saturation threshold of complex, compute-heavy applications compared to simple, low-latency microservices where the communication cost is the primary bottleneck. However, introducing a physical network fundamentally alters this capacity. In the Social Network 2 Host deployment, the saturation point occurs significantly earlier at 1000 RPS. In the Calculator 2 Host deployment we see saturation occurring at 1250 RPS as well. This indicates that the presence of a physical network in the service mesh introduces severe bottlenecks that substantially reduce the overall saturation point and throughput capacity of the application.

5.2 Bare-Metal vs. Virtualized Environments

As observed in Sections 4.1.1 and 4.2.1, the Virtual Machines deployment consistently exhibited the highest latency among the four configurations, with one notable exception in the high-load Social Network scenario. While both the 2 hosts and virtual machines deployments distribute the application across two nodes connected by a physical network, the virtual machine configuration introduces a hypervisor layer and virtual networking stack that would impact performance characteristics.

Table 8: Social Network service processing latency at 100 and 1000 RPS for 2 hosts and virtual machines deployment.

Deployment	Load (RPS)	Mean Latency (ms)	P95 Latency (ms)	P99 Latency (ms)	% Change of P99 Latency
2 hosts	100	7.76	19.71	27.18	0
virtual machines	100	22.5	52.9	66.47	144.55
2 hosts	1000	275.85	1188.78	1724.4	0
virtual machines	1000	365.82	916.7	1575.28	-8.64

Table 9: Calculator service processing latency at 100 and 1000 RPS for 2 hosts and virtual machines deployment.

Deployment	Load (RPS)	Mean Latency (ms)	P95 Latency (ms)	P99 Latency (ms)	% Change of P99 Latency
2 hosts	100	8.06	20.84	24.21	0.00
virtual machines	100	64.52	114.08	134.13	454.03
2 hosts	1000	24.87	189.51	257.73	0.00
virtual machines	1000	1905.05	2783.48	2993.6	1061.53

Table 8 and 9 provide an updated summary of the 2 hosts and virtual machines latency measurements for Social Network and Calculator. In the Social Network application at 100 RPS, virtualization results in a 144.55% increase in P99 service processing latency compared to the bare-metal 2 Host setup. However, at 1000 RPS, the virtual machine deployment actually shows an 8.64% decrease in P99 latency and a decrease in P95 latency compared to the 2 Host deployment, despite the Mean latency increasing by 32.62%. This suggests that while virtualization adds a constant overhead to the average request, the hypervisor's resource scheduling may actually limit the most extreme tail latency spikes under heavy load. With the enforcing scheduling policies, this can prevent the uncontrolled resource contention seen on bare-metal systems, which results in a reduction in tail latencies.

The Calculator application demonstrates a much more straightforward relationship with virtualization, where the VM deployment increases P99 service processing latency by 454.03% at 100 RPS and 1061.53% at 1000 RPS. Because the Calculator services perform minimal computation, the relative impact of the virtualization stack and virtual network bridge is significantly amplified. Interestingly, the absolute latency measurements for the Calculator at 100 RPS are higher than those of the Social Network (134.13 ms vs 66.47 ms). This is counter-intuitive given the Calculator's simpler design and suggests a possible internal performance bottleneck within the Calculator service's implementation.

5.3 Optimized Distributed Deployments

In Section 4.3, we explore a distributed deployment strategy that optimizes connections within the service mesh to reduce latency. This configuration, denoted as the 2 host hybrid deployment, utilizes UDS for services co-located on the same host and reverts to TCP for communication between services residing on separate physical hosts. By leveraging UDS for local traffic, this approach seeks to mitigate the protocol overhead typical of loopback TCP connections.

At low loads of 100 RPS, this optimization provides negligible benefits. The P99 service processing latency remains nearly identical between the 2 host regular and 2 host hybrid deployments, with differences of less than 1 ms. Because resources are not yet constrained, the base network traversal cost of the inter-host TCP links remains the dominant latency factor, and the reduction in local communication overhead is not significant enough to impact the overall tail latency.

However, at higher loads of 1000 RPS, the 2 host hybrid deployment demonstrates a clear advantage in managing tail latency. Notably, the P99 service processing latency is 8.02% lower compared to the 2 host regular deployment. This suggests that as the system approaches saturation, the increased efficiency of UDS helps reduce the total processing and queuing burden on individual services, preventing some of the latency amplification seen in distributed deplotment that use only TCP connections in the service mesh.

Despite these improvements in latency, the saturation profile of the application remains largely unchanged. As observed in the latency-versus-load curves, saturation for the hybrid configuration still occurs at 1250 RPS, a result similar to the 2 host regular deployment. This indicates that while the optimization lowers the latency at high loads, it does not significantly increase the application's total load capacity, as the physical network link between the two hosts remains the ultimate bottleneck.

Furthermore, while this optimization proves beneficial for high-load latency, it introduces significant operational restrictions. First, this configuration requires each service to be manually or programmatically configured to identify which protocol to use when contacting specific downstream services. Second, utilizing UDS for inter-service communication creates a rigid

coupling; once two services are configured to use UDS, they are forced to remain co-located on the same physical host. Any redeployment that separates these services would require a reconfiguration of their connection to TCP, increasing the complexity of dynamic scaling and orchestration in a microservice environment.

5.4 Limitations

While this research provides a systematic characterization of communication overheads, several limitations must be acknowledged that may affect the generalizability of the results.

First, the 1 host UDS deployment for the Social Network application does not utilize UDS for every inter-service connection. Due to the complexity of the Social Network architecture, a portion of the service mesh continued to utilize TCP for communications between services. Consequently, the latency measurements for this configuration likely capture some TCP protocol overheads, meaning the recorded performance represents a conservative estimate of the potential latency reduction achievable through a pure UDS implementation.

Second, the Calculator application may contain internal implementation bottlenecks. These architectural constraints likely contributed to the unexpectedly high absolute latency measurements recorded at low loads, values that occasionally exceeded those of the more complex Social Network application. In these instances, the observed latency is a product of both communication overhead and service-level performance limitations, making it difficult to isolate the networking cost entirely.

Finally, the resource allocation for the Virtual Machines deployment introduced a discrepancy in available compute power. The virtual machines were configured to utilize only half of the physical hardware resources (CPU and memory) available to the bare-metal 2 Host deployment. This reduction in available resources likely made computational constraints and queuing delays more apparent in the virtualized environment, potentially overstating the overheads in virtualization.

6.Future Work

The findings of this study provide a foundation for several avenues of future research aimed at further deconstructing microservice communication overheads.

First, it is essential to extend this experimental framework to a broader range of microservice benchmark suites beyond the Social Network and Calculator applications. Evaluating diverse architectural patterns will determine if the performance characteristics and TCP overheads identified here remain consistent across different application paradigms.

The unexpected performance seen in Virtual Machines warrants deeper investigation. Future experiments should involve re-provisioning the virtual environment with compute resources

identical to the physical hosts to eliminate computational resource constraints as a variable. This would allow for a more improved measurement of the virtualization overhead impact on tail latency.

Finally, future research should explore the scalability and adaptability of the hybrid deployment model across more complex environments. This includes implementing the UDS and TCP hybrid strategy within the Social Network application to determine if the performance gains observed in simpler services are amplified in a complex architecture. Additionally, further investigation is required to test variations of the hybrid configuration for the Calculator application, such as distributing services across three or more physical hosts or within a virtualized cluster. Evaluating these expanded topologies would provide a more comprehensive understanding of how hybrid communication meshes perform as the degree of distribution and the complexity of the underlying infrastructure increase.

7. Conclusion

This study demonstrates the communication overheads inherent in microservice architectures by comparing TCP and UDS across varying deployment configurations. We demonstrate that TCP incurs significant overhead compared to UDS, and that this effect is substantially more pronounced in simpler applications like the Calculator application, where the cost of traversing the network stack accounts for a larger portion of the total latency budget. Furthermore, the results show that distributed deployments across physical networks and virtualized environments incur massive additional overheads due to network stack traversal and virtualization.

Finally, we show that implementing a hybrid communication model, utilizing a mix of UDS for local intra-host traffic and TCP for inter-host requests, can successfully reduce tail latencies at high loads by approximately 8.02% compared to traditional all-TCP distributed deployments. While these local optimizations improve latency performance, the overall saturation point remains constrained by the physical network.

8. References

[1] P. D. Francesco, I. Malavolta and P. Lago, "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption," 2017 IEEE International Conference on Software Architecture (ICSA), Gothenburg, Sweden, 2017, pp. 21-30, doi: 10.1109/ICSA.2017.24.

[2] A. Elfatatry, "Microservices: A Review of the Costs and the Benefits," in Proc. 5th Int. Conf. Advances and Trends in Software Engineering (SOFTENG 2019), Valencia, Spain, 2019, pp. 23–27.

[3] A. Detti, L. Funari and L. Petrucci, "μBench: An Open-Source Factory of Benchmark Microservice Applications," in IEEE Transactions on Parallel and Distributed Systems, vol. 34, no. 3, pp. 968-980, 1 March 2023, doi: 10.1109/TPDS.2023.3236447.

[4] S. Kurpad, S. BT, S. Vijaykumar, S. Jain and S. Kalambur, "Microarchitectural Analysis and Characterization of Performance Overheads in Service Meshes with Kubernetes," *2023 3rd Asian Conference on Innovation in Technology (ASIANCON)*, Ravet IN, India, 2023, pp. 1-6, doi: 10.1109/ASIANCON58793.2023.10270428.

[5] F. Yashu, S. Malhotra, and M. Saqib, "Optimizing Intra-Container Communication with Memory Protection Keys: A Novel Approach to Secure and Efficient Microservice Interaction," May 2025, arXiv:2505.07836. [Online]. Available: https://arxiv.org/abs/2505.07836

[6] Y. Gan et al., "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in Proc. 24th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), New York, NY, USA, 2019, pp. 3–18. doi: 10.1145/3297858.3304013.

[7] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann and S. Kounev, "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research," 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Milwaukee, WI, USA, 2018, pp. 223-236, doi: 10.1109/MASCOTS.2018.00030.

[8] UBC Systopia, "uServices-Benchmarks," 2024. [Source code]. Available: https://github.com/ubc-systopia/uservices-benchmarks.

[9] Docker Inc., "Docker," version 27.x, 2024. [Online]. Available: https://www.docker.com. [Accessed: Sept. 5, 2025].

[10] A. A. Krasnovsky, "Evaluating Asynchronous Semantics in Trace-Discovered Resilience Models: A Case Study on the OpenTelemetry Demo," Dec. 2025, arXiv:2512.12314. [Online]. Available: https://arxiv.org/abs/2512.12314