# On the Measurement of io_uring Performance: a Strategy and Experience in the Envoy Service Mesh

**Lucas O. Martínez**
DEI - Universidad Católica "Nuestra Señora de la Asunción"
Asunción, Paraguay
*lucasoctaviom@gmail.com*

and

**Raúl Gutierrez**
DEI - Universidad Católica "Nuestra Señora de la Asunción"
Asunción, Paraguay
*rgs@hey.com*

and

**Luca Cernuzzi**
DEI - Universidad Católica "Nuestra Señora de la Asunción"
Asunción, Paraguay
*lcernuzz@uc.edu.py*
ORCID: 0000-0001-7803-1067

**Abstract**

A growing number of projects from various areas have recently incorporated support for io_uring, a novel technique to reduce system calls and accelerate processing capabilities in network services. This new approach promises to yield benefits within the context of modern infrastructure patterns, such as those embracing the service mesh paradigm. Among these projects, Envoy Proxy has been in the process of gaining io_uring support. An initial experimental trial using Envoy with io_uring reports a 10% increase in bandwidth along with a 10% reduction in latency. Unfortunately there is scarce companion data to support the results and the corresponding communication protocols that were used, all of which would have been quite useful to facilitate the reproducibility of these experiences. In order to make technical details more explicit, to facilitate reproducibility of measurement experimental studies, and to compare experimental results, the main goal of the present study aims to measure and systematise the impact of io_uring in an Envoy-based service mesh using our proposed matrix of operations and protocols. Additionally, our motivation includes enhancing the performance of service meshes, which are critical components in modern distributed systems architectures. Therefore, we have also designed a measurement strategy which is composed of key metrics to assess the impact of io_uring in service mesh. Our test results show improvements reducing latency and increasing bandwidth. More specifically, for HTTP/1 the obtained results are on par with the initial claims whereas for HTTP/2 they are better than the previous study.

**Keywords:** Service Mesh, Envoy, io_uring, Performance.

## 1 Introduction and Motivation

Traditional input/output operations – even when we account for variations that use asynchronous techniques – exhibit a known set of limitations [1] [2]. The reliance on this kind of limitations, specifically within certain contexts such as those denominated as web scale scenarios, can be the main driver for suboptimal performance [3]. As an attempt to escape from these limitations, io_uring was proposed in 2019 [4][5].

Within the context of the advances heralded by io_uring, it's worthwhile to note that the service mesh paradigm [6][7][8] has continued to gain traction during these recent years consolidating its place and utility in modern infrastructures [9]. Consequently multiple projects – which also implement the service mesh interface – have embraced io_uring, as has been the case with Envoy Proxy[1]. The development and support for io_uring within Envoy[2] began in December 2021. By April 8 of 2022 the first commits related to this feature landed in Envoy's main branch. It's important to mention that the integration of io_uring into Envoy Proxy goes beyond supporting a new API. This implementation aligns the capabilities of io_uring with the specific needs of Envoy, optimizing IO patterns for better performance. Despite not being considered complete nor stable yet, it's arguable that a crucial aspect to facilitate further adoption is quantifying the performance improvements.

Regarding the measurement of performance of io_uring in service mesh, a relevant previous experimental study that we established as a starting point is the initial set of pull requests to add io_uring support into Envoy [10]. These proposed code changes claim that by reducing the number of necessary system calls using io_uring, a 10% increase in bandwidth and a 10% decrease in latency can be achieved [10]. While the described results are significant and relevant within an industrial context, several questions remain unaddressed. Indeed, the documentation does not specify whether the improvement is restricted to a subset of the vast available set of functionality within Envoy or if they apply universally. Therefore, a primary contribution of this study aims to measure and formalise the potential impact of io_uring carefully noting the different types of functionalities and the protocols that are used when performing the measurement. We adopted Envoy for our study because existing performance evaluations of io_uring in service meshes have exclusively involved Envoy so far.

Within this study, we aim to establish a bridge between the world of performance measurements with synthetic traffic and the more ad-hoc practices in the industry. The motivation here is supported by the fact that while there has been work in this area [11][12][13][14], there are currently no standards for measuring performance in service meshes. Therefore, as an additional and potentially more expansive contribution, we propose a strategy for measuring performance within a service mesh.

In consequence, during this work we will be focusing on two main research questions; the first one (Main-RQ-1), and more comprehensive, focusing on what strategy to use for measuring performance within a service mesh, and second one (Main-RQ-2), the more specific about the performance impact of using io_uring in an Envoy service mesh.

This work represents an extension of the work presented at CLEI 2023 [15] in two main dimensions. Firstly, it expands the literature review and, consequently, the section related to related works presenting the protocol of a Multivocal Literature Review (MLR) conducted and broadening the search to gray literature that provides richer and more detailed information. Secondly, it analyses in detail the experimental results regarding the HTTP/2 protocol which echoes the findings previously obtained with the HTTP/1 protocol and – in a way – generalises the experimental results obtained.

In the tests with both protocols, the results are positive and consistent with those of the previous study [10]. Moreover, with the HTTP/2 protocol the experimental results are better than those obtained previously.

Our expectation is that the findings of this work can be of value for researchers and industry practitioners interested in the service mesh paradigm, as well as for the broader microservices architecture community.

The rest of the study is structured as follows: in Section 2, we provide a brief overview of io_uring; in Section 3, we discuss related works; in Section 4 we present the measurement strategy and the design and implementation of the service mesh to test the performance; in Section 5 we focus on the comparison and evaluation of the performance of the service mesh with and without io_uring; finally, in Section 6 we present the conclusion.

## 2 io_uring: a new paradigm for input/output operations

One of the fundamental shortcomings of traditional system calls for input/output operations – such as *read(2)* and *write(2)* along with their corresponding vectored variants – is that they are strictly synchronous interfaces. By definition, this means that the system call will return only when a complete response is received or an error occurs along the process [1]. This approach is bound by a strict computational cost, given that it involves copying information during context switches (between user space and the kernel, and vice versa) for each invocation. It is important to note that even though benign at first look, the copying of bytes from one space to another is what dominates the costs associated with input and output in a network.

There are asynchronous alternatives to these input/output calls, commonly referred to as *aio* variants (asynchronous input/output). They do, however, come along with significant limitations [2]. On one hand, the interfaces they expose present significant challenges when trying to use them correctly. In certain cases, they require

[1] https://www.envoyproxy.io/

[2] https://envoy.com/

dealing with the specific semantics of different signals that may or may not be reentrant [16]; the reentrant ones being even more subtle in their complexity [17]. On the other hand, a signal can be safely used with asynchronous calls if it can be reissued from a signal handler [17]. These signals can be considered unsafe if the actions of a signal handler can interfere with the operation being interrupted [18]. Moreover, the *aio* family doesn't entirely eliminate the cost incurred during context switches that comes with each system call. In other words, it does not avoid the tax that emanates from copying information between user space and the kernel (and vice versa) through the lifecycle of a process.

In an effort to overcome some of these problems, io_uring was proposed in 2019 [4][5]. Its fundamental design idea revolves around using two ring buffers (queues), one dedicated to pending requests (the SQ or Submission Queue) with another dedicated to completed requests (the CQ or Completion Queue). This new input/output technique comes with several advantages. One of them relies on the fact that its buffer queues are "shared" between user space and the kernel, thus eliminating the need to spend time copying requests and responses between the two contexts (becoming effectively a zero-copy technique [19][20]). To further reduce redundant computation, all access validations are performed at setup time. Another superior aspect of io_uring over the *aio* system calls family is that the latter only works for files that can be accessed with the O_DIRECT mode. This limitation essentially restricts the use of *aio* to databases or storage systems [21]. Whereas on the other hand, io_uring works with virtually all IO modes including cached files, direct access files and – potentially the most relevant use case – sockets. The sum of all these features have driven the rise in usage of io_uring during the last few years.

Despite its compelling advantages, io_uring cannot escape the fact that it is a relatively new technique when we take into consideration the amount of time required for systems programming interfaces to mature. So it is only natural to expect that its implementation will require ongoing corrective maintenance. Furthermore, given that io_uring is implemented in kernel space and not user space, newly discovered vulnerabilities pose a high risk of destabilising the entire operating system. Additionally, the implementation of io_uring in Linux is written in C which is more susceptible to certain types of exploits [22][23] than higher-level languages as for example might be the case with Rust programming language which recently gained support in the Linux kernel [24][25]. Therefore the initial set of bugs reported for io_uring have been related to memory handling errors such as buffer overflows, dangling pointers, and read/write of uninitialized data [26].

As of late 2023, io_uring in Envoy [27] is still in a state of continuous change due to unresolved design and implementation details. One of the most notable current limitations is the incomplete support for using the HTTPS protocol, we will elaborate further on this in Section 5.4. Indeed, the currently functional protocols are HTTP and HTTP/2.

Considering these factors, and to support the adoption of this new paradigm, it is necessary to conduct experimental evaluations demonstrating the performance benefits of io_uring. Specifically, the main focus of this study is to analyze its performance within a service mesh environment. Therefore, the next section focuses on the measurement of performance in a service mesh.

## 3 Related Works

With the objective of determining the most commonly used methods and metrics for evaluating performance in a service mesh, we have conducted a Multivocal Literature Review (MLR). Firstly, we have adapted and applied the systematic review process proposed by Kitchenham et al. [28] to a study focused on performance measurements in service meshes.

**Table 1:** Research Questions for the Multivocal Literature Review Study

| Research Questions (RQ) |
|---|
| **RQ-1:** What are the most commonly used Layer 7 protocols for performance measurement in a service mesh? |
| **RQ-2:** What are the most considered metrics when evaluating performance in a service mesh? |
| **RQ-3:** How is performance typically measured in a service mesh? |
| **RQ-4:** What tools are used to assist during the performance testing and subsequent analysis? |
| **RQ-5:** What technique or techniques are commonly used to generate traffic in a service mesh? |
| **RQ-6:** What are the characteristics of the test environment used for testing? |

Since the popularity of service meshes is relatively recent – less than 10 years – we have decided not to restrict the search to a specific time period. The selection made regarding the main, alternative, or related terms is presented in Table 2.

**Table 2:** Search String Terms

| Main terms | Alternative or related terms |
|---|---|
| Quality | quality OR consistency OR maintainability OR metric OR measure OR testability OR scalability |
| Service Mesh | service mesh OR envoy OR istio OR linkerd |
| Performance | performance OR latency OR bandwidth usage OR efficiency OR traffic |

The search string defined based on Table 2 is as follows:

("quality" OR "consistency" OR "maintainability" OR "metric" OR "measure" OR "testability" OR "scalability") AND ("service mesh" OR "envoy" OR "istio" OR "linkerd") AND ("performance" OR "latency" OR "bandwidth usage" OR "efficiency" OR "traffic")

### 3.1 First Approach

Regarding the sources, it should be noted that we have worked with papers or research available on Google Scholar, ResearchGate, IEEE, or CICCO[3]. In particular, CICCO is the digital search portal offered by CONACYT in Paraguay, including libraries such as ACM, Scopus, Web of Science, Nature, Science Direct, and various relevant publishers. While including numerous sources along with Google Scholar may appear redundant given its extensive reach, it's crucial to acknowledge that, in our experience, Google's algorithm may not surface all relevant documents. Therefore, conducting explicit searches in the additional sources could uncover new and valuable insights.

Also, with the idea of staying true to the systematic review process, we initially decided to work only with sources from qualified literature, excluding gray literature.

**Table 3:** Inclusion and Exclusion Criteria (first approach)

| Inclusion criteria | Exclusion criteria |
|---|---|
| + Books and papers with tests and empirical results<br>+ In case multiple papers report very similar studies, the most recent and with a better background will be selected | - Studies that do not present empirical results |

The result of the search has yielded the following studies:

- "Impact of etcd deployment on Kubernetes, Istio, and application performance", Lars Larsson et al. [29]
- "An Empirical Study of Service Mesh Traffic Management Policies for Microservices", Mohammad R. et al. [30].

As expected, given that the topic is relatively new, we have found only few studies containing relevant information for our research, which is very limited. Therefore, in order to obtain more relevant and representative results, we have decided to expand the search to also include gray literature.

### 3.2 Second Approach

Although some insightful information can be found in research, the issue lies more in the sphere of practitioners. Hence, to obtain relevant results, it would not be advisable to exclude gray literature. For this reason, we expanded our review study. Similarly, being a relatively new field, we have decided to minimise the exclusion criterias in order to find a larger number of articles, or literatures, related to the proposed research questions. Additionally, while the search string remained unchanged during this approach, we expanded our sources. We retained the same sources as in the first approach (see Section 3.1) but also included Google search and GitHub to find gray literature.

[3] https://cicco.conacyt.gov.py/

**Table 4:** Inclusion and Exclusion Criteria (second approach)

| Inclusion criteria | Exclusion criteria |
|---|---|
| + Books, papers, articles, blogs, official documentations, Git repositories with tests, and empirical results | - Studies that do not present empirical results |
| + In case multiple papers report very similar studies, the most recent and with a better background will be selected | - Literature available only in form of summary |
| | - Literature available exclusively in PowerPoint presentation format |

In addition to the previously used sources (Google Scholar, ResearchGate, CICCO, and IEEE), the same search string has also been applied for searching official project documentation, such as Envoy, as well as blogs that provide empirical evidence.

The result of the search for gray literature has yielded the following studies:

- "Evaluating Performance And Security Characteristics Of Service Mesh Technologies In A Rancher 2.X Environment", Marius Jøsok Nesset [31]
- "Latency-aware Optimization of the Existing Service Mesh in Edge Computing Environment", Zhen Sun [32]
- "Testing Resilience of Envoy Service Proxy with Microservices", 2019, Nikhil Dattatreya Nadig [33]
- "Best Practices: Benchmarking Service Mesh Performance", Megan O'Keefe et al. [34]
- "Performance and Scalability", by Istio [35]
- "Benchmarking Linkerd and Istio", William Morgan [36]
- "Linkerd Benchmarks", William Morgan [37]
- " Performance Benchmark Analysis of Istio and Linkerd", Thilo Fromm [11]
- "Benchmarking Envoy Proxy, HAProxy, and NGINX Performance on Kubernetes", Richard Li [38]
- "Benchmarking 5 Popular Load Balancers: Nginx, HAProxy, Envoy, Traefik, and ALB", Gerred Dillon [39]

### 3.3 Discussion of Related Works

While there are studies related to measuring the performance of a service mesh, these are not abundant. The first noticeable observation is that there were few academic articles found in the literature, while, as expected for the applied and novel theme, several studies come from industrial environments and gray literature. Here are the main insights from the literature regarding the metrics and measurements most commonly used when evaluating performance in a service mesh. We provide a specific analysis regarding the research questions presented earlier.

RQ-1: What are the most commonly used Layer 7 protocols for performance measurement in a service mesh? The most frequently used protocol when measuring performance in a service mesh is HTTP/1.1 [11][29-39], appearing in 100% of the articles studied throughout our literature review. This was expected as HTTP is likely the most widely used protocol on the network. The second most popular was gRPC [31][33][36-37], appearing in the 33% of the articles, indicating a possible trend towards newer protocols when working with service meshes. However, io_uring is not yet available for the gRPC protocol in the Envoy service mesh. It is important to note that HTTPS was mentioned in only one article [39], which may lead us to assume that researchers might be implicitly considering HTTPS as part of HTTP, or they are excluding it due to the encryption it performs, assuming that this, despite carrying an extra load, will not affect the results when evaluating HTTP. Due to this, it could be interesting, in future studies, to assess the performance in a service mesh using both HTTP and HTTPS, and then compare them to verify if the performance of one is reflected or extrapolated to the other.

RQ-2: What are the most considered metrics when evaluating performance in a service mesh? Mainly, latency [11][31-38], CPU usage [31][32][34-37], and memory usage [31][34-37] are the most commonly used metrics when measuring performance in service meshes. Latency stands out from the other two, being used as a metric in 75% of the investigated studies, while CPU and memory usage in 50% and 41.7% of the studies respectively. Regarding CPU and memory usage, we have noticed ambiguity with respect to the context, meaning that it varies depending on the author of the study or research. But why do we say it is ambiguous? Most studies that evaluated the mesh with

synthetic traffic measured CPU and/or memory usage in relation to a predetermined number of requests per second and took the maximum measurement – with respect to each of these metrics – as the base for the study. Alternatively, one could assess the average usage of these resources. This latter way of evaluating resource consumption is more commonly seen in production environments. Thus, we observe that the ambiguity lies in defining how to measure the consumption of these resources, so we can assert that there is no consensus regarding this. Despite this lack of consensus, it is noteworthy that these three metrics – latency, CPU usage, and memory usage – are usually used together.

RQ-3: How is performance typically measured in a service mesh? The use of Requests Per Second (RPS) is the common technique used when measuring performance in a service mesh [11][29][31-39]. This has been used in almost 92% of the investigated articles. The RPS metric is used to evaluate performance in the service mesh when it is subjected to a specific number of requests per second, which can be translated into a stress test. But what is a request? Is it always the same? We can define a request as a message transmitted from the client to the server. The size of this message usually varies from service to service; this implies that we cannot ensure that the size of the requests used in different studies are the same. That being said, when dealing with synthetic traffic, the difference between the sizes of the messages should not be large enough to alter the results. However, what about using traffic from the production environment? Here, the “request factor” comes into play, which is ambiguous and dependent on the application or service. Nevertheless, whether working with synthetic or production traffic, it would be useful to have a set of predefined "packet sizes" to use during performance tests to facilitate contrasting the performance of one technique or another, in a service mesh. Similarly, it is not clear for how long the stress test with RPS should be conducted: 30 seconds?, one minute?, or until the system collapses? Undoubtedly, there is a need for better standardization of methods related to measuring performance in a service mesh.

RQ-4: What tools are used to assist during the performance testing and subsequent analysis? We have observed that there is no established preference regarding the tools used to assist during the performance testing of a service mesh. This is probably due to the fact that service meshes are still relatively new, and practitioners are still "finding their ideal tool". Having mentioned this, it might be interesting to reconsider this question in the near future. We can categorize these tools into two groups: load generators and monitoring or benchmarking tools. Load-generating tools were used in almost 67% of the studies. Among the studies that mention using a load-generating tool, none stands out as a favorite, considering each study uses a different one. The same behaviour has been observed regarding monitoring tools. Prometheus, Grafana, and Kivolk seem to be the most popular, each appearing in only 16% of the articles, hence we cannot consider it a determining factor. It is worth mentioning that Prometheus and Grafana were used together on each occasion.

RQ-5: What technique or techniques are commonly used to generate traffic in a service mesh? The most common way is through a synthetic traffic generator [11][30-39]. Its popularity could be due to the ease of use, as obtaining "real" or production data in a considerable quantity can be quite challenging for individuals or small organizations. Additionally, we also believe that tests conducted with production traffic – or a replica of it – were carried out privately by organizations to measure their systems in a more precise manner, not sharing them with the community due to privacy concerns. Considering this, the overwhelming preference for synthetic traffic could be a "false positive." Nevertheless, given the community's greater ease of access to synthetic traffic generators, this should continue to be the preferred method. Despite synthetic traffic being the favorite method for the industry, it is evident that when measuring performance it is more convenient to have traffic similar to that of production environments to obtain more meaningful results.

PI-6: What are the characteristics of the test environment used for testing? Regarding the service mesh itself, Istio is currently the most widely used [11][29-34], followed by Linkerd [11][31][36-37], both highly popular in projects found on GitHub. Additionally, the cloud – given its expansion and scalability properties – is the preferred environment for deploying a service mesh [11][29][32][33][36-39]. Details such as memory, operating system, processor type, and other more specific data are not often mentioned in the studies and research, making it challenging to make fully refined comparisons.

## 4 Measuring io_uring performance

In this section, the performance measurement strategy for the service mesh is defined, including the metrics to be considered and the design of the test system. Additionally, essential concepts related to the configuration of Envoy Proxy are discussed to facilitate understanding when analysing the configuration of the test service mesh used.

### 4.1 Measurement Strategy

To answer our first main research question (Main-RQ-1) on how to measure performance in a service mesh, we propose to consider the following measurement strategy:

- Measurement of CPU and memory consumption;
- Latency measurement (including percentiles) for:
    - Successful responses, meaning those with status code 200 to 299;
    - Total processing, without discriminating the status code of the responses. It is worth mentioning that if this is to be reproduced in an industrial or production environment, it is always advisable to filter to evaluate only requests with successful responses. The concept of "total processing" is usually defined in two ways: the first, as the process that occurs from sending the first byte of the request to receiving the last byte of the response; the second, encompasses the time from when the request is prepared, before the first byte is sent, to the moment the connection is closed because the response was received. Both cases have been considered in our measurements;
    - Queueing and setup, which basically refers to the process of connecting with the server;
- Bandwidth measurement (consumption).

### 4.2 Design and Implementation

To perform the performance measurement in a service mesh, we propose focusing on three main components:

- the service mesh
- the client
- the server.

For the first component – the service mesh – we worked with Envoy, which is our initial motivation for this research due to its recent implementation of io_uring. Regarding the version used, we have worked with a set of patches that are not currently in the main branch. There are two ways to work with the latest implementation of io_uring (in Envoy). The first is to merge the changes from the pull request into the main branch (or the branch you want to work on). The second alternative is to clone the fork of the person/people responsible for implementing io_uring support in Envoy. We opted for the first alternative, considering it to be more complete (or stable) than those found in forks.

For the client, the second component, we worked with Nighthawk, an L7 (HTTP, HTTPS, HTTP/2) performance characterization tool [40]. We have chosen Nighthawk for the following reasons:

- It features a load testing client supporting HTTP/1.1 and HTTP/2 over HTTP and HTTPS;
- It includes a binary to transform Nighthawk's output into well-known formats (such as JSON and a Fortio[4]-ready format), allowing integration with other systems and dashboards;
- It has native integration for visualising data in Fortio; and,
- It is integrated with Envoy, eliminating the need for additional dependencies.

The third and final main component is the service's server. Since we are using Nighthawk for the client, we have decided to also use it for the server. Nighthawk, besides providing a client and a benchmark tool, also provides a test server. This test server is not a traditional HTTP server but an extension of Envoy whose logic runs as an HTTP filter.

Thus, the Nighthawk Test Server will be the sole service for the measurement system. We have configured the test server to respond to any request with a body of 10 bytes and the header {foo: bar2}. Additionally, we have configured the server without any artificial latency injection to obtain a more accurate performance measurement

In summary, we have designed a test system consisting of the Nighthawk client, Envoy as proxy, and the Nighthawk Test Server as an Envoy filter (see Figure 1).

---

[4] https://fortio.org/

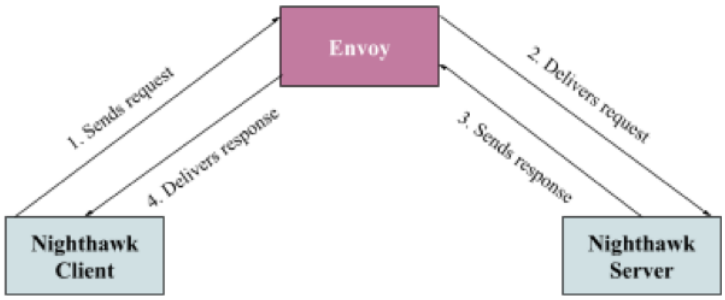


**Figure 1**: Diagram of the test system used for the evaluation, consisting of the Nighthawk client, Envoy, and the Nighthawk server as the sole backend service

To avoid uncontrolled variations caused by shared CPU resources – since all components will be running on the same machine – and to obtain more accurate measurements, we ran each component using the Linux *taskset* command. This command allows us to execute a task with a specific CPU affinity. CPU affinity is a scheduler property that assigns a process to a particular set of the system CPUs. It is a mechanism to assign a process to a CPU and prevent the process from being executed on other CPUs [41], thus avoiding migration costs. It is important to clarify that this does not mean exclusive access to that CPU; other mechanisms, such as *cgroups* (control groups), are used for that purpose.

Regarding these measurements, we expect CPU and memory consumption to be mostly constant with minimal fluctuations. Also, concerning latency, a similar behaviour is expected, with low variations in the higher percentiles.

As for the client, it will perform 1,000 requests per second for 60 seconds, resulting in a total of 60,000 requests. These requests are of a fixed size (20 megabytes), which could limit the representativeness of the diversity and distribution of requests found in real-world usage. Ergo, for future work, exploring variations in sizes could simulate real-world conditions more faithfully and achieve better alignment with practical usage scenarios.

In summary, for the devised strategy, we have a total of five essential steps to conduct performance tests in the service mesh:

1. Deploying the Nighthawk server;
2. Deploying the Envoy proxy with io_uring;
3. Running the script responsible for executing the client and registering resource-consumption measurements;
4. Transforming the outputs to CSV format, for performance comparison;
5. Repeating steps 1 to 4, with the only difference being that step 2 is performed using the Envoy configuration without io_uring.

We have decided to conduct the performance tests in a local environment due to its easy accessibility. The tests were performed on a system with an Intel i5-7600K processor, 16 GB of RAM, and Ubuntu 20.04.4 LTS (Focal Fossa) as the operating system.

## 5 Analysis of the Results

With the test system and performance measurement strategy defined, we proceed to data collection, evaluation, and comparison. We will analyze the results of the performance measurement of the service mesh in its base configuration and explore the measurement with io_uring handling the input and output of the mesh. We will first evaluate the performance over HTTP, and finally over HTTP/2.

### 5.1 Results Representation

For the evaluated protocols (HTTP and HTTP/2), we will distinguish the following:

1. Graphs for resource consumption (CPU and memory);
2. Tables with the results of the measurements by the Nighthawk client;
3. Bandwidth graph.

It is worth mentioning that we registered the bandwidth measurement in KB/s. We also have graphs of RPS (requests per second) that help visualize the distribution of percentiles.

It is important to clarify that the comparison of the results obtained across different tests could be subject to slight fluctuations due to various factors influencing the CPU and memory consumption of the service mesh [42]. Moreover, the test environment is not entirely hermetic. It is also known that the number of connected proxies influences the performance [35].

We will now consider the metrics mentioned in Section 4.1, which are directly related to the following four tables that we will present later on:

- Benchmark_http_client.latency_2xx
  - Contains data related to the latency of requests with successful responses (Response Status Code 200-299).
- Queueing and connection setup latency
  - Contains data on latency concerning queuing and connection with the server. In other words, the time it takes for the request to be served [43][44].
- Request start to response end
  - Total latency from the start of the request until the server's response is received. Similar to Benchmark_http_client.latency_2xx but including responses with status codes other than 2xx.
- Initiation to completion
  - Total time elapsed from initiation to completion of the query. It is always somewhat greater than the data reported in Request start to response end, as additional milestones such as request preparation are taken into account.

Each table is followed by a graph with the empirical Cumulative Distribution Function (eCDF) to illustrate the percentile-latency relationship in a more user-friendly way.

It is important to note that the definitions for Benchmark_http_client.latency_2xx, Queueing and connection setup latency, Request start to response end, and Initiation to completion are not universal. Therefore, they can be interpreted differently, causing serious difficulties for those encountering them in multiple studies. Such ambiguities are not exclusive to these terms; for example, Varnish[5] inverts the definitions of call timers with timers for sockets that are in an idle state [7]. Another situation highlighting the lack of a universal language is found when comparing the standard metrics of Istio [45] with those used by Nighthawk.

As mentioned in Section 4, the test client is responsible for making 1,000 requests per second for 60 seconds, resulting in a total of 60,000 requests. We analyze latency with respect to percentiles because average values can be deceptive due to their simple and one-dimensional nature. To illustrate this limitation, let's consider an update to a video streaming service. If its average latency remains relatively unchanged, we might think that the update did not affect the service quality. However, by examining percentiles, if we notice a significant increase in latency for the 80th or 90th percentile, it would reveal a performance decline caused by this update. Having said this, it's worth mentioning that the average is usually heavily influenced by the 90th percentile, so such a scenario is unlikely, but it serves its purpose in arguing for the use of percentiles for analysis.

Recalling that the performance measurements are conducted for HTTP and HTTP/2 protocols, we will now proceed with the analysis of the behaviour of io_uring in an Envoy service mesh. Each comparison table consists of three parts:

1. The performance of Base Envoy (without io_uring).
2. The performance of Envoy integrated with io_uring.
3. The percentage variation of latency. It is important to note that this variation is in relation to Envoy integrated with io_uring, being positive if its performance is better (lower latency) than that of its base version. Otherwise, this value is negative.

[5] https://varnish-cache.org/

With the tables defined, we can proceed to describe the columns and headers of each:

- General data or macro (in microseconds):
  - Min: minimum latency of the population;
  - Mean: mean latency of the population;
  - Max: maximum latency of the population;
  - Pstdev: standard deviation of the population;
- Specific data per percentile:
  - Percentile: the percentile being referenced;
  - Value (microseconds): maximum latency for the related percentile, measured in microseconds;
  - Variation: percentage representing the variation in maximum latency between base Envoy and its version with io_uring. As mentioned before, this value is in relation to the latency of the base Envoy implementation, i.e., a positive value indicates that the latency has been reduced by that proportion. The calculation is given by the following formula:

$$(L_B - L_{IO}) / L_B$$

Where $L_B$ represents the latency of the base implementation, and $L_{IO}$ the latency of the implementation with io_uring.

### 5.2 Performance with HTTP

Let's start by analyzing resource consumption. For HTTP, both implementations of the service mesh, base Envoy and its counterpart with io_uring, show similar resource consumption. Given this fact, we cannot confidently assert the superiority in terms of resource usage for either of the two approaches. Nevertheless, we will analyze the differences with more details.

On one hand, we observe that the service mesh in its base state consumes an average of 19.72% of the CPU (see Figure 2), while the implementation that integrates io_uring consumes 20.52% (see Figure 3). This would indicate that CPU consumption increases by 0.8% with the integration of io_uring.

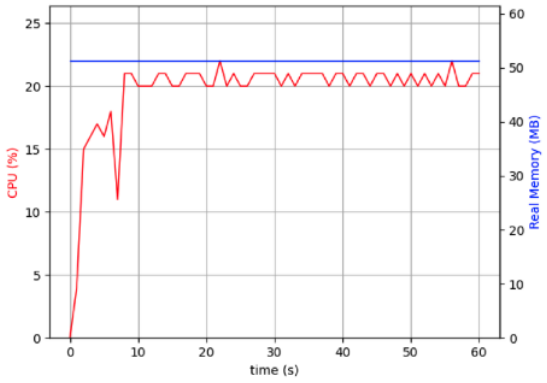


**Figure 2**: CPU (%) and memory (MB) usage during the 60-second test for the HTTP protocol without io_uring

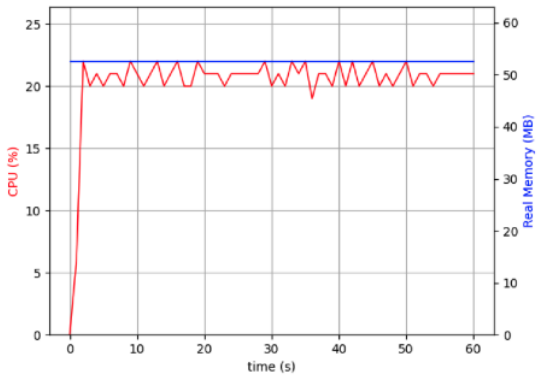


**Figure 3**: CPU (%) and memory (MB) usage during the 60-second test for the HTTP protocol with io_uring

Regarding bandwidth, comparing the computed averages, we have 462.60 KB/s for Envoy with io_uring, while the base implementation is at 415.17 KB/s. With these values, we can observe that the io_uring implementation results in a 10.26% increase in bandwidth usage (see Figure 4).

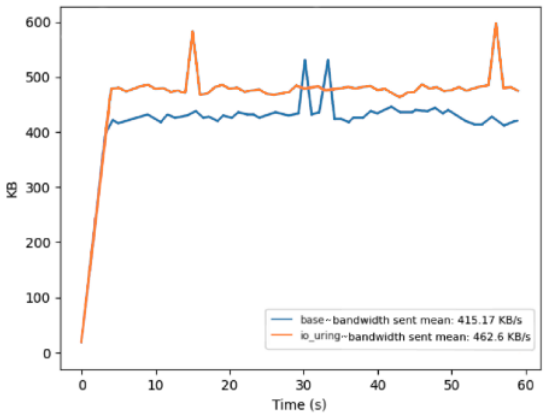


**Figure 4**: Bandwidth usage registered for Envoy with io_uring (orange) and without io_uring (blue), with the HTTP protocol

*5.2.1 Benchmark_http_client.latency_2xx for HTTP*

It is always convenient to start by comparing the difference between the average latencies to get a basic idea of the performance difference between both implementations. On one hand, the average latency for benchmark_http_client.latency_2xx in base Envoy is 1,044 microseconds, while for its counterpart with io_uring, it is 859 microseconds. This implies a 17.72% performance improvement in favor of io_uring.

**Table 5:** Macro/General comparison between base Envoy and Envoy with io_uring, regarding benchmark_http_client.latency_2xx, with the HTTP protocol

| | **Latency measurements in microseconds** | | | |
|---|---|---|---|---|
| | **Min** | **Mean** | **Max** | **Pstdev** |
| **Base Envoy** | 517 | 1,044 | 26,119 | 1,288 |
| **Envoy w/ io_uring** | 520 | 859 | 38,899 | 1,017 |

**Table 6:** Percentile comparison between base Envoy and Envoy with io_uring, regarding benchmark_http_client.latency_2xx, using the HTTP protocol

| Latency | Base Envoy | Envoy w/ io_uring | |
|---|---|---|---|
| **Percentile** | **Value (µs)** | **Value (µs)** | **Variation** |
| 0.50 | 676 | 680 | -0.59% |
| 0.75 | 697 | 688 | 1.29% |
| 0.80 | 739 | 693 | 6.22% |
| 0.90 | 1,915 | 778 | 59.37% |
| 0.95 | 2,905 | 1,741 | 40.07% |
| 0.990 | 7,039 | 5,813 | 17.42% |
| 0.9990 | 14,468 | 13,022 | 9.99% |

In a more detailed analysis, considering the percentiles, for the 50th percentile we observe that io_uring results in a 0.59% performance decrease in the service mesh, which might be considered not relevant given its proximity to zero. However, reaching the 80th percentile, a significant difference becomes noticeable: a 6.22% improvement when using io_uring. The most remarkable change is observed at the 90th percentile, with a 59.37% improvement in performance (i.e., lower latency) with io_uring. The 95th percentile reveals a quite interesting and significant latency difference, with io_uring showing a 40.07% lower latency. Although falling in the 95th percentile and beyond is challenging, they have been included to contrast with the results of the study that motivated this work [10].

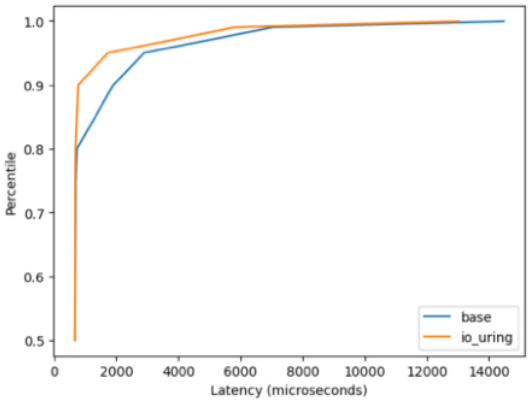


**Figure 5**: Percentile-latency relationship for base Envoy and Envoy with io_uring, regarding benchmark_http_client.latency_2xx, using the HTTP protocol

*5.2.2 Queueing and Connection Setup Latency for HTTP*

For Queueing and Connection Setup Latency (QCSL), the average latency (Table 7) in our io_uring-integrated service mesh was 33 microseconds, while in the base version of the service mesh, it was 35 microseconds. This indicates, on average, that io_uring manages to increase efficiency by 5.17% for the queuing and connection setup action, considering the latency time.

**Table 7:** Macro/General comparison between base Envoy and Envoy with io_uring, regarding Queueing and Connection Setup Latency (QCSL), with the HTTP protocol

| | QCSL measurements in microseconds | | | |
|---|---|---|---|---|
| | **Min** | **Mean** | **Max** | **Pstdev** |
| **Base Envoy** | 27 | 35 | 3,227 | 31 |
| **Envoy w/ io_uring** | 27 | 33 | 974 | 7 |

**Table 8:** Percentile comparison between base Envoy and Envoy with io_uring, regarding Queueing and Connection Setup Latency (QCSL), using the HTTP protocol

| QCSL Latency | Base Envoy | Envoy w/ io_uring | |
|---|---|---|---|
| **Percentile** | **Value (µs)** | **Value (µs)** | **Variation** |
| 0.50 | 32 | 32 | 0.00% |
| 0.75 | 33 | 33 | 0.00% |
| 0.80 | 33 | 33 | 0.00% |
| 0.90 | 38 | 34 | 10.53% |
| 0.95 | 44 | 39 | 11.36% |
| 0.990 | 68 | 54 | 20.59% |
| 0.9990 | 160 | 79 | 50.63% |

When analyzing the percentiles in detail (Table 8), we notice that there is no variation, at least significantly enough to be captured, until reaching the 90th percentile, where io_uring improves performance by 10.53%. In other words, we observe less striking or more conservative measures compared to the data evaluated for Table 6.

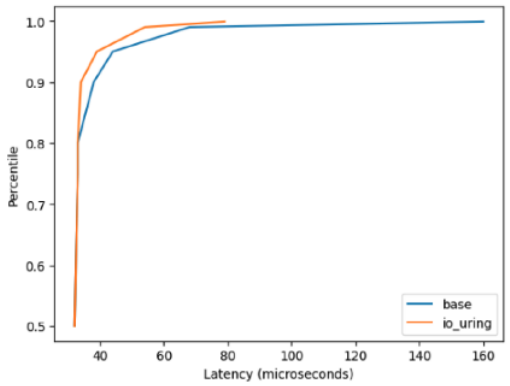


**Figure 6**: Percentile-latency relationship for base Envoy and Envoy with io_uring, regarding Queueing and Connection Setup Latency, using the HTTP protocol

*5.2.3 Request Start to Response End for HTTP*

As reflected in Table 9, once again – now for Request Start to Response End (RSRE) – the service mesh integrated with io_uring improves the performance, on average, compared to the base version of the mesh.

**Table 9:** Macro/General comparison between base Envoy and Envoy with io_uring, regarding Request Start to Response End (RSRE), with the HTTP protocol

| | RSRE measurements in microseconds | | | |
|---|---|---|---|---|
| | **Min** | **Mean** | **Max** | **Pstdev** |
| **Base Envoy** | 517 | 1,040 | 26,119 | 1,288 |
| **Envoy w/ io_uring** | 520 | 859 | 38,899 | 1,017 |

**Table 10:** Percentile comparison between base Envoy and Envoy with io_uring, regarding Request Start to Response End (RSRE), using the HTTP protocol

| RSRE Latency | Base Envoy | Envoy w/ io_uring | |
|---|---|---|---|
| **Percentile** | **Value (µs)** | **Value (µs)** | **Variation** |
| 0.50 | 676 | 679 | -0.44% |
| 0.75 | 697 | 688 | 1.29% |
| 0.80 | 739 | 692 | 6.36% |
| 0.90 | 1,914 | 777 | 59.40% |
| 0.95 | 2,905 | 1,740 | 40.10% |
| 0.990 | 7,039 | 5,812 | 17.43% |
| 0.9990 | 14,466 | 13,021 | 9.99% |

By taking a look at the percentiles, Table 10, we notice the results are quite similar to the 2xx latency table (benchmark_http_client.latency_2xx - Table 6), with the difference that it includes not only successful requests but all the requests. Given that the test system is entirely in a local environment, we can expect that almost 100% of the requests result in successful responses. This is reflected in the values of this table, which are practically identical to the values measured exclusively for successful responses. Even the difference between the means of the latencies registered for successful responses (Table 6) and RSRE (Table 9) does not exceed one microsecond.

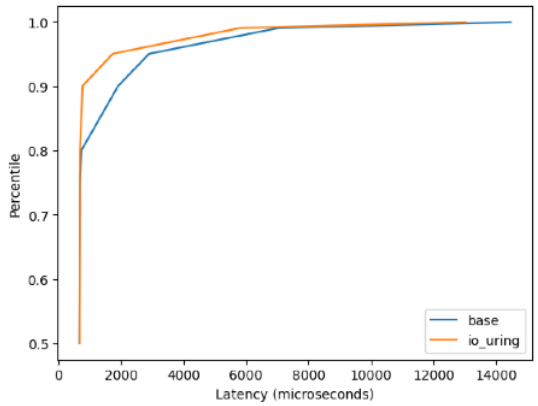


**Figure 7**: Percentile-latency relationship for base Envoy and Envoy with io_uring, regarding Request Start to Response End, using the HTTP protocol

*5.2.4 Initiation to Completion for HTTP*

Finally, we analyze the Initiation to Completion table. Since it contains data regarding latency from initiation to completion, including the time spent on assembling or setting up the request, it is evident that it will display the longest times (latency) seen so far. Likewise, for reasons similar to those detailed in the discussion of Table 10, the values behave similar to those on Table 6. Moreover, it would be more accurate to say that the tables of "benchmark_http_client.latency_2xx" and "Request start to response end" are similar to this one.

**Table 11:** Macro/General comparison between base Envoy and Envoy with io_uring, regarding Initiation to Completion (IC), with the HTTP protocol

| | IC measurements in microseconds | | | |
|---|---|---|---|---|
| | **Min** | **Mean** | **Max** | **Pstdev** |
| **Base Envoy** | 569 | 1,107 | 26,167 | 1,309 |
| **Envoy w/ io_uring** | 572 | 916 | 38,955 | 1,025 |

**Table 12:** Percentile comparison between base Envoy and Envoy with io_uring, regarding Initiation to Completion (IC), using the HTTP protocol

| IC Latency | Base Envoy | Envoy w/ io_uring | |
|---|---|---|---|
| **Percentile** | **Value (µs)** | **Value (µs)** | **Variation** |
| 0.50 | 729 | 732 | -0.41% |
| 0.75 | 751 | 742 | 1.20% |
| 0.80 | 801 | 747 | 6.74% |
| 0.90 | 2,025 | 841 | 58.47% |
| 0.95 | 3,030 | 1,844 | 39.14% |
| 0.990 | 7,175 | 5,887 | 17.95% |
| 0.9990 | 15,105 | 13,112 | 13.19% |

Regardless, by analyzing Table 11 we find that the average latency of the mesh integrated with io_uring (916 microseconds) is 17.25% lower – that is, more efficient – than that obtained in the base Envoy mesh (1,107 microseconds). Also, if we analyze the 90th percentile, we see a performance, by io_uring, 58.47% better than that of its base counterpart. Again, showing similarities with the previously mentioned tables.

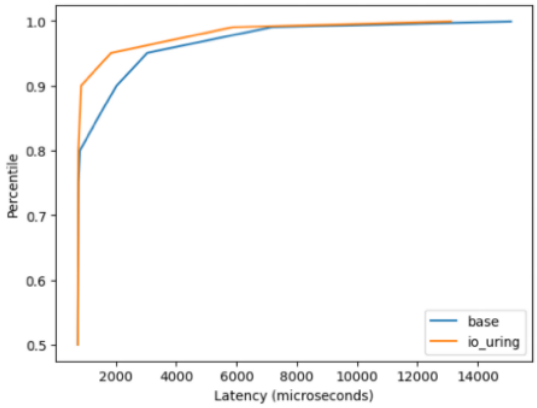


**Figure 8**: Percentile-latency relationship for base Envoy and Envoy with io_uring, regarding Initiation to Completion, using the HTTP protocol

### 5.3 Performance with HTTP/2

For HTTP/2, the implementation of the service mesh with base Envoy and its counterpart with io_uring exhibit a behaviour that we could consider similar in terms of resource consumption. On one hand, the base implementation uses, on average, 21.14% of the CPU, while its counterpart with io_uring shows an average use of 22.83%, indicating an increase of 1.69% in CPU usage by io_uring. This difference may seem insignificant to some due to its small magnitude; however, this behaviour is worthy of note.

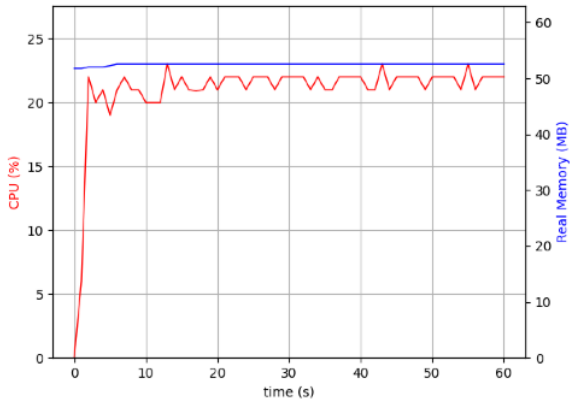


**Figure 9**: CPU (%) and memory (MB) usage during the 60-second test for the HTTP/2 protocol without io_uring

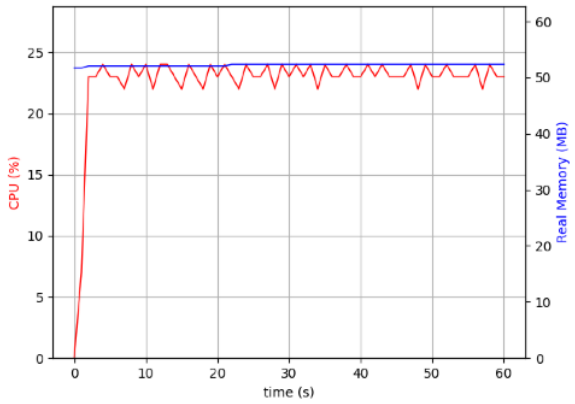


**Figure 10**: CPU (%) and memory (MB) usage during the 60-second test for the HTTP/2 protocol with io_uring

Regarding bandwidth, we observed average values of 311.16 KB/s and 358.74 KB/s for base Envoy and Envoy with io_uring respectively, indicating that the implementation of io_uring results in a 13.27% higher bandwidth usage.

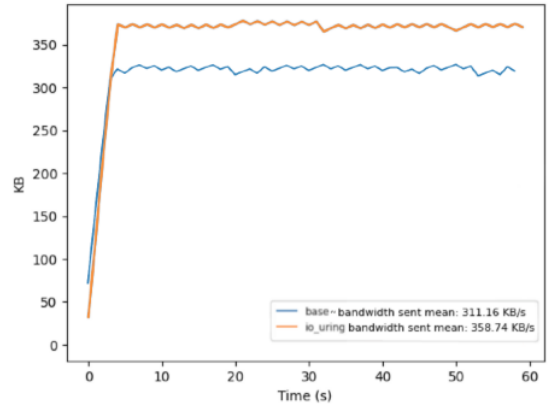


**Figure 11**: Bandwidth usage registered for Envoy with io_uring (orange) and without io_uring (blue), with the HTTP/2 protocol

*5.3.1 Benchmark_http_client.latency_2xx for HTTP/2*

Let's start by analyzing the difference between the average latencies registered for the population of successful responses, or also called benchmark_http_client.latency_2xx (Table 13). The average latency registered in our service mesh with base Envoy was of 935 microseconds – considerably less than the one recorded for HTTP/1, as expected from HTTP/2 – compared to the 767 microseconds of its counterpart with integrated io_uring. This implies a performance improvement of 17.97%, on average, when integrating io_uring to Envoy.

**Table 13:** Macro/General comparison between base Envoy and Envoy with io_uring, regarding benchmark_http_client.latency_2xx, with the HTTP/2 protocol

| | Latency measurements in microseconds | | | |
|---|---|---|---|---|
| | **Min** | **Mean** | **Max** | **Pstdev** |
| **Base Envoy** | 537 | 935 | 36,257 | 1,278 |
| **Envoy w/ io_uring** | 544 | 767 | 15,539 | 452 |

**Table 14:** Percentile comparison between base Envoy and Envoy with io_uring, regarding benchmark_http_client.latency_2xx, using the HTTP/2 protocol

| **Latency** | **Base Envoy** | **Envoy w/ io_uring** | |
|---|---|---|---|
| **Percentile** | **Value (µs)** | **Value (µs)** | **Variation** |
| 0.50 | 715 | 717 | -0.28% |
| 0.75 | 739 | 733 | 0.81% |
| 0.80 | 749 | 738 | 1.47% |
| 0.90 | 853 | 757 | 11.25% |
| 0.95 | 1,461 | 814 | 44.28% |
| 0.990 | 6,458 | 1,984 | 69.28% |
| 0.9990 | 19,610 | 7,899 | 59.72% |

If we take a look at the equivalent table for HTTP/1 (Table 6, see section 5.2.1)), we will notice a similar pattern: an almost negligible decrease in performance at the 50th percentile when using io_uring, and an improved performance from the 75th percentile onwards. For the 90th percentile, we observe that the implementation integrating io_uring has a performance 11.25% better than its counterpart. Regarding the higher percentiles, we see an incredible performance increase with io_uring, but as mentioned during the analysis of Table 6, falling within these high percentiles is quite challenging, so it should not be our main focus. That said, it is still important to appreciate the significant advantage that io_uring has in these high percentiles.

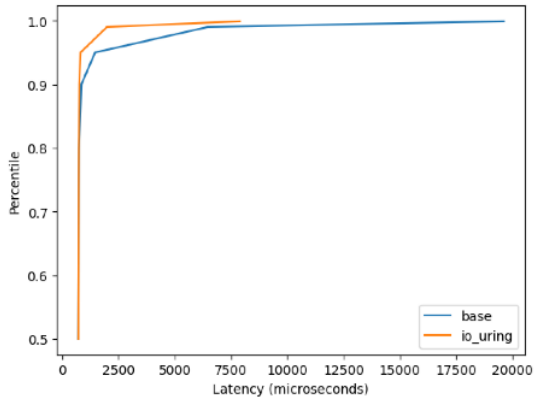


**Figure 12**: Percentile-latency relationship for base Envoy and Envoy with io_uring, regarding benchmark_http_client.latency_2xx, using the HTTP/2 protocol

*5.3.2 Queueing and Connection Setup Latency for HTTP/2*

Our next comparatives tables evaluate the Queueing and Connection Setup Latency (QCSL). Once again, we continue to notice a similar behaviour to what was observed during the analysis for the HTTP/1 protocol: differences in latency occur from the 90th percentile onwards. In that percentile, io_uring maintains the advantage by a 6.45% difference.

**Table 15:** Macro/General comparison between base Envoy and Envoy with io_uring, regarding Queueing and Connection Setup Latency (QCSL), with the HTTP/2 protocol

| | QCSL measurements in microseconds | | | |
|---|---|---|---|---|
| | **Min** | **Mean** | **Max** | **Pstdev** |
| **Base Envoy** | 47 | 58 | 2,684 | 19 |
| **Envoy w/ io_uring** | 47 | 56 | 2,695 | 11 |

**Table 16:** Percentile comparison between base Envoy and Envoy with io_uring, regarding Queueing and Connection Setup Latency (QCSL), using the HTTP/2 protocol

| **QCSL Latency** | **Base Envoy** | **Envoy w/ io_uring** | |
|---|---|---|---|
| **Percentile** | **Value (µs)** | **Value (µs)** | **Variation** |
| 0.50 | 55 | 55 | 0.00% |
| 0.75 | 57 | 57 | 0.00% |
| 0.80 | 57 | 57 | 0.00% |
| 0.90 | 60 | 58 | 6.45% |
| 0.95 | 70 | 60 | 14.29% |
| 0.990 | 104 | 69 | 33.65% |
| 0.9990 | 146 | 86 | 41.10% |

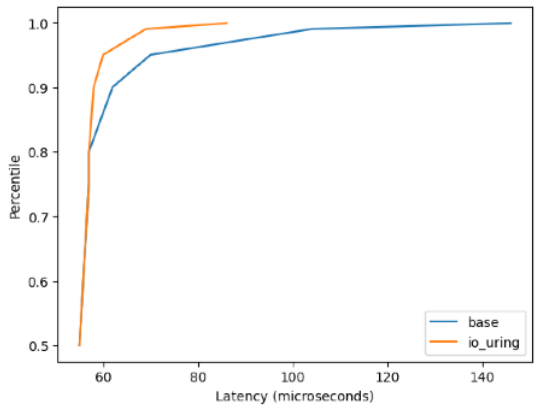


**Figure 13**: Percentile-latency relationship for base Envoy and Envoy with io_uring, regarding Queueing and Connection Setup Latency, using the HTTP/2 protocol

*5.3.3 Request Start to Response End for HTTP/2*

Coming almost to the end of the comparative tables, we analyze the Request Start to Response End (RSRE). As mentioned earlier in the discussion of Table 10 – Request Start to Response End for HTTP/1 (see section 5.2.3) – the similarity between Benchmark_http_client.latency_2xx and Request Start to Response End is evident at first glance. The difference lies in the fact that the latter not only includes successful requests, but all requests. By comparing the average latency we observe that the performance using io_uring (766 microseconds) is 17.99% better than its base version of Envoy (934 microseconds). Also, by looking at the 90th percentile we can appreciate a 11.37% latency improvement in favor of our service mesh with integrated io_uring.

**Table 17:** Macro/General comparison between base Envoy and Envoy with io_uring, regarding Request Start to Response End (RSRE), with the HTTP/2 protocol

| | RSRE measurements in microseconds | | | |
|---|---|---|---|---|
| | **Min** | **Mean** | **Max** | **Pstdev** |
| **Base Envoy** | 537 | 934 | 36,255 | 1,278 |
| **Envoy w/ io_uring** | 544 | 766 | 15,539 | 452 |

**Table 18:** Percentile comparison between base Envoy and Envoy with io_uring, regarding Request Start to Response End (RSRE), using the HTTP/2 protocol

| **RSRE Latency** | **Base Envoy** | **Envoy w/ io_uring** | |
|---|---|---|---|
| **Percentile** | **Value (µs)** | **Value (µs)** | **Variation** |
| 0.50 | 714 | 717 | -0.42% |
| 0.75 | 739 | 733 | 0.81% |
| 0.80 | 748 | 738 | 1.34% |
| 0.90 | 853 | 756 | 11.37% |
| 0.95 | 1,461 | 814 | 44.28% |
| 0.990 | 6,458 | 1,983 | 69.29% |
| 0.9990 | 19,610 | 7,899 | 59.72% |

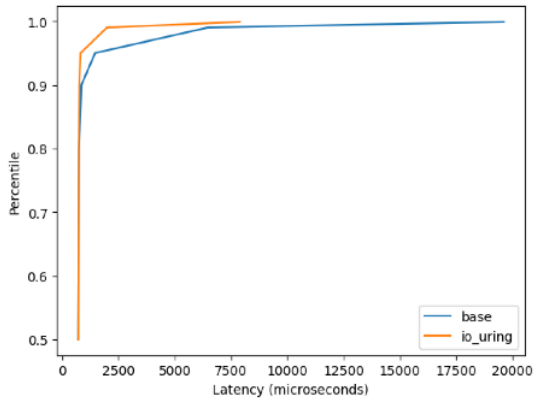


**Figure 14**: Percentile-latency relationship for base Envoy and Envoy with io_uring, regarding Request Start to Response End, using the HTTP/2 protocol

*5.3.4 Initiation to Completion for HTTP/2*

Finally, we analyze the performance of HTTP/2 regarding Initiation to Completion (IC). The following tables demonstrate a similar behaviour to the previous ones, as expected. With an average latency of 1,017 microseconds in our service mesh with base Envoy and 844 microseconds in the mesh with integrated io_uring, hence resulting in a 17.01% better average performance by using io_uring. Regarding percentiles, we have similar results as those presented in Table 18, with an 11.53% improvement using io_uring.

**Table 19:** Macro/General comparison between base Envoy and Envoy with io_uring, regarding Initiation to Completion (IC), with the HTTP/2 protocol

| | IC measurements in microseconds | | | |
|---|---|---|---|---|
| | **Min** | **Mean** | **Max** | **Pstdev** |
| **Base Envoy** | 611 | 1,017 | 36,341 | 1,292 |
| **Envoy w/ io_uring** | 617 | 844 | 15,625 | 454 |

**Table 20:** Percentile comparison between base Envoy and Envoy with io_uring, regarding Initiation to Completion (IC), using the HTTP/2 protocol

| IC Latency | Base Envoy | Envoy w/ io_uring | |
|---|---|---|---|
| **Percentile** | **Value (µs)** | **Value (µs)** | **Variation** |
| 0.50 | 791 | 793 | -0.25% |
| 0.75 | 819 | 812 | 0.85% |
| 0.80 | 829 | 817 | 1.45% |
| 0.90 | 945 | 836 | 11.53% |
| 0.95 | 1,570 | 897 | 42.87% |
| 0.990 | 6,600 | 2,060 | 68.79% |
| 0.9990 | 19,793 | 7,970 | 59.73% |

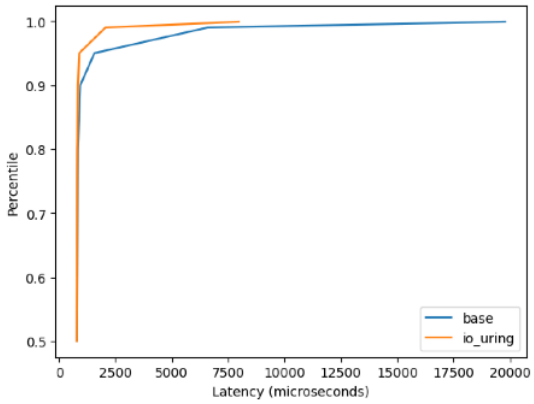


**Figure 15**: Percentile-latency relationship for base Envoy and Envoy with io_uring, regarding Initiation to Completion, using the HTTP/2 protocol

### 5.4 Threats to Validity

A general threat to validity is the current immaturity of the io_uring implementation in Envoy. As the implementation evolves, optimizations, refactoring, and other modifications may lead to variations in performance. Regardless, the proposed measurement strategy remains a valuable foundation for conducting new tests and measuring performance as the technology evolves.

Another relevant aspect is the support for the HTTPS protocol with io_uring in Envoy, at the time of writing, is not finalized despite this protocol being one of the most widely used on the network. This limitation might pose a significant constraint on the generalizability of our findings, as the majority of real-world applications rely heavily on HTTPS for secure communications. This lack of support means that our current evaluation primarily focuses on non-secure HTTP protocols, which might not accurately represent the performance characteristics in a fully secured environment.

Moreover, one of the primary concerns is the assumption that performance gains observed with HTTP will translate similarly to HTTPS once support is implemented. This assumption may not hold true due to the additional computational overhead associated with encryption and decryption processes inherent to HTTPS. Additionally, our evaluation environment and test scenarios cannot capture all the complexities and variabilities of production environments. Factors such as network topology, varying load patterns, and the presence of other concurrent processes could influence performance in ways not fully accounted for in our tests.

It should be noted that once io_uring fully supports HTTPS, the experiments conducted will maintain a certain level of replicability not only in terms of the measurement strategy but also regarding the architecture used for testing. Additionally, it's worth mentioning that using real application data instead of simulated data, if possible and available, could enhance the reliability of the results.

By acknowledging these limitations and clearly stating the conditions under which our results are valid, we aim to provide a more transparent and reliable assessment of io_uring's impact on Envoy-based service meshes.

### 5.5 General Considerations and Discussion

Next, we discuss and emphasize the key findings of the measurements and their respective comparisons, addressing our second main research question (Main-RQ-2) about the performance impact of using io_uring in an Envoy service mesh. Perhaps the most interesting metric is the latency of successful responses (benchmark_http_client.latency_2xx - Tables 6 and 14) since they are, in theory, the most common cases. In this case, for both HTTP and HTTP/2, there are no significant benefits until the 75th percentile. In fact, at the 50th percentile, io_uring resulted in a very slight performance decline: -0.59% and -0.28% for HTTP and HTTP/2 respectively. It is in the higher percentiles where we notice the benefit io_uring offers. For the 90th percentile and above, the performance difference is quite surprising, reaching an improvement – in the conducted tests – of up to 59.37% for HTTP and 69.28% for HTTP/2. Similarly, on the highest percentile, the 99.9th, we observe that io_uring achieves a latency improvement of 9.99% for HTTP, confirming the initial hypothesis about io_uring for Envoy [10], and a more significant improvement of 59.72% with HTTP/2. The complete process – Request Start to Response End and Initiation to Completion – presents similar results due to the fact that successful responses are part of these sets. Moreover, being in a local and controlled environment makes it highly likely that all requests result in successful responses. That said, we recognize the importance of extending our evaluation beyond the controlled local test environment to cover more general scenarios, ensuring the broader applicability of our findings. While our local focus ensured consistent successes, addressing potential results of failed requests in dynamic network contexts will enrich our understanding on the subject. As part of future work, a quantitative analysis of the latency gains facilitated by the integration of io_uring compared to latencies related to the network will provide a refined understanding of its effectiveness.

Regarding Queueing and Connection Setup Latency, we found similar results. In the early percentiles – 50th, 75th, and 80th – there are no variations compared to their counterparts without io_uring; i.e. they remain exactly the same in terms of latency. As we reach higher percentiles, meaning 90th to 99.9th, the performance improvements in favor of io_uring become noticeable, moreover, these improvements become more significant as we approach the upper percentile, ranging from 6.45% to 50.63% between these percentiles (see Tables 16 and 8 respectively).

Based on our tests, we confirmed that io_uring has a positive impact on performance in both the HTTP and HTTP/2 protocols, causing fascinating improvements concentrated in the higher percentiles. On the other hand, io_uring is slightly less efficient in the 50th percentile, which is an interesting finding and requires further investigation. This behaviour is counterintuitive, as one might expect that as more requests come in, the slower the system should become. So, why does io_uring behave the way it does? One possible hypothesis is that this behaviour is due to the cost that io_uring pays upon initialization, i.e., all the configurations it performs when initialized. These configurations could be the cause of its lower performance in the lower percentiles. But what about the improvement in the higher percentiles? Continuing with the hypothesis, after completing all the necessary initial configurations, due to the functioning of io_uring, fewer system calls occur, resulting in the observed improvement. In simple terms: io_uring pays a slight price upon initialization, to later achieve considerable improvements.

This positive impact is not only evident in latency but also in bandwidth usage. For the HTTP protocol, we observed a 10.26% increase in bandwidth usage when using io_uring in the service mesh. This value is quite close to that reported by the Intel team responsible for developing io_uring for Envoy [10]. Meanwhile, with HTTP/2, the result is slightly higher, resulting in a 13.27% increase when using io_uring.

The experimental results obtained for both protocols, HTTP and HTTP/2, suggest that the improvements provided by io_uring could be independent of the protocol used. However, for a more comprehensive understanding of the results, it would be necessary to extend the study, especially to the HTTPS protocol. Nevertheless, as previously mentioned, the development of io_uring for Envoy is still incomplete, hence these functionalities can be experimentally verified as future advancements unfold.

Regarding a potentially negative aspect, we observed a slightly higher CPU usage when using io_uring; specifically, 0.8% with HTTP and 1.69% with HTTP/2. We conducted several additional tests to verify if this measurement is consistent and reproducible. In all tests, CPU consumption by io_uring was consistently between 0.5% and 2% higher than the base implementation. Regarding this increase in CPU usage, it would be useful to observe CPU usage in conjunction with the requests per second for a more precise measurement and more insightful knowledge. As for future works, it would be interesting to conduct a comprehensive analysis, possibly using eBPF (Extended Berkeley Packet Filter), to understand the cause of this unexpected behaviour. It's essential to note that higher CPU usage is not necessarily a regression, but may be the natural cost of achieving higher bandwidth usage and lower latency. Another important point to mention is that the tests did not reveal significant differences related to memory usage, both for RSS (Resident Set Size) and VSZ (Virtual Memory Size) memory. The observed behaviour might be due to the fact that we did not perform queries of significant sizes to force memory usage.

Lastly, the potential negative impact of io_uring on certain metrics could be associated with the current level of maturity of this technology. It is expected that as novel technologies like io_uring evolve in complex software environments, their performance may also improve.

## 6 Conclusion

The rise of the io_uring interface presents a promising solution to the historical problems associated with traditional input/output system calls and their related alternatives like asynchronous input/output (*aio*). As observed throughout this study, the performance improvement achieved by using the io_uring subsystem benefits service meshes which inherently incur an additional performance cost by virtue of being a new layer of abstraction. As demonstrated in Section 5, adopting io_uring yields several dividends. Specifically, we observe a 9.99% better latency performance for HTTP and an even more remarkable 59.72% improvement for HTTP/2. Similar rewards are evident in bandwidth, with a 10.26% increase for HTTP using io_uring along with a 13.27% increase for HTTP/2. Thus this interface could be fundamental to overcome the previously mentioned performance disadvantages, upholding the adoption of service meshes which are still seeing a rise in their adoption and demand.

From the perspective of software engineering, the results of io_uring are more than encouraging for both the academic community and the industry. These results encourage the exploration of alternatives to interfaces that have long been established within the ecosystem of kernel and systems programming.

A crucial factor in advocating for the adoption of io_uring is having access to empirical evidence of the related performance improvements. This aligns with the main focus of this study, demonstrating how the integration of io_uring can reduce latency, increment bandwidth and overall improve the performance in Envoy based service meshes, addressing our second main research question (Main-RQ-2).

Another contribution of this study has focused on proposing a performance measurement strategy for service meshes. This constitutes a first step towards conducting more systematic and replicable experiments, directly addressing our first main research question (Main-RQ-1) about what strategy to use for measuring performance within a service mesh. Over time, this can consolidate experimental evidence and consequently analyse the advantages and limitations of service meshes in a more rigorous and precise manner.

In more general terms, it would be desirable to move towards greater formalisation or standardisation in the service mesh paradigm to aid in the definition of universal protocols for measurements or benchmarking. This would benefit both the industry with its implementation practices as well as academia with its methods and analysis.

**References**


[1] Z. Bin, J. Qing Chao, Y. Ling Xiao and C. Xiao Guang, "Research and limitation of system call based on Linux platform," in 2011 International Conference on Electric Information and Control Engineering, Wuhan, China, 2011, pp. 1592-1595, https://ieeexplore.ieee.org/document/5777226

[2] Z. Brown (Oracle), "Asynchronous System Calls", 2007 - https://www.kernel.org/doc/ols/2007/ols2007v1-pages-81-86.pdf

[3] R. Norris, "io_uring is not an event system", 2021 - https://despairlabs.com/posts/2021-06-16-io-uring-is-not-an-event-system/

[4] Jens Axboe, "Efficient IO with io_uring", 2019 - https://kernel.dk/io_uring.pdf

[5] A. De Sarker, "Getting Hands on with io_uring using Go", 2020 - https://developers.mattermost.com/blog/hands-on-iouring-go/

[6] W. Li, Y. Lemieux, J. Gao, Z. Zhao and Y. Han, "Service Mesh: Challenges, State of the Art, and Future Research Opportunities," in 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), 2019, pp. 122-1225, https://ieeexplore.ieee.org/document/8705911

[7] R. Gutiérrez Segalés, 2021, "Implementando una Malla de Servicios", tésis de grado, DEI - Universidad Católica "Nuestra Señora de la Asunción"

[8] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," in IEEE Software, vol. 35, no. 3, pp. 24-35, May/June 2018, https://ieeexplore.ieee.org/document/8354433

[9] Z. Jory, "The Top 3 Service Mesh Developments in 2020", 2019 - https://thenewstack.io/the-top-3-service-mesh-developments-in-2020

[10] D. Rozhkov (Intel) Contribution to Envoy, “io: add iohandle backed with io_uring” - https://github.com/envoyproxy/envoy/pull/19082

[11] T. Fromm, “Performance Benchmark Analysis of Istio and Linkerd”, 2019 - https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd/

[12] L. Calcote, M. Ganguli, S. Ranganath and O. Van Der Schaaf, “Analyzing Service Mesh Performance”, 2021 - https://layer5.io/resources/service-mesh-performance/analyzing-service-mesh-performance

[13] M. Kipper, “Benchmarking Istio & Linkerd CPU”, 2019 - https://medium.com/@michael_87395/benchmarking-istio-linkerd-cpu-c36287e32781

[14] I. Sim, “Linkerd 2.0 and Istio Performance Benchmark”, 2018 - https://ihcsim.medium.com/linkerd-2-0-and-istio-performance-benchmark-df290101c2bb

[15] L. O. Martínez, R. Gutierrez and L. Cernuzzi, "A Measurement Strategy for io_uring Performance in the Envoy Service Mesh", 2023 XLIX Latin American Computer Conference (CLEI), La Paz, Bolivia, 2023, pp. 1-10, https://ieeexplore.ieee.org/document/10346093

[16] D. McCall, “Asynchronous I/O and event notification on linux”, 2017 - http://davmac.org/davpage/linux/async-io.html

[17] “Reentrancy, thread safe, asynchronous signal safe, interrupt safe and cancellation safe”, 2020 - https://sites.google.com/site/embeddedmonologue/home/c-programming/reentrancy-thread-safe-asynchronous-signal-safe-interrupt-safe-and-cancellation-safe

[18] Oracle Documentation, “Signal Handlers and Async-Signal Safety” - https://docs.oracle.com/cd/E19455-01/806-5257/gen-26/index.html

[19] S. Hussain, “What is io_uring?”, 2020 - https://unixism.net/loti/what_is_io_uring.html

[20] T. Endo and S.M. Saleh Al-Mashni, “Comparative Evaluation of Asynchronous IO Interface between io_uring and libaio implemented in a NoSQL DB for SSD”, 2019 - https://jglobal.jst.go.jp/en/detail?JGLOBAL_ID=202002249376629901

[21] G. Costa, “How io_uring and eBPF Will Revolutionize Programming in Linux”, April2021 - https://thenewstack.io/how-io_uring-and-ebpf-will-revolutionize-programming-in-linux/

[22] E.D. Berger, C. Hollenbeck, P. Maj, O. Vitek and J. Vitek, “On the Impact of Programming Languages on Code Quality”, in ACM Transactions on Programming Languages and Systems, 2019, pp 1–24, https://dl.acm.org/doi/10.1145/3340571

[23] S. Nanz, C. A. Furia, “A comparative study of programming languages in Rosetta code”, in ICSE '15: Proceedings of the 37th International Conference on Software Engineering, 2015, pp. 778–788, https://dl.acm.org/doi/10.5555/2818754.2818848

[24] J. Iry, “Types à la Chart”, 2010 - http://james-iry.blogspot.com/2010/05/types-la-chart.html

[25] A. Gaynor, “Introduction to Memory Unsafety for VPs of Engineering”, 2019 - https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering/

[26] E. D. Berger and B. G. Zorn, “DieHard: probabilistic memory safety for unsafe languages”, in ACM SIGPLAN Notices, 2006, pp. 158-168, https://dl.acm.org/doi/abs/10.1145/1133255.1134000

[27] D. Rozhkov (Intel) Contribution to Envoy, “Migrate event loops from file events to buffer events” - https://github.com/envoyproxy/envoy/issues/17922

[28] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, S. Linkman, “Systematic literature reviews in software engineering – A systematic literature review”, Information and Software Technology, Volume 51, Issue 1, pp. 7-15, 2009, https://doi.org/10.1016/j.infsof.2010.03.006

[29] L. Larsson, W. Tärneberg, C. Klein, E. Elmroth and M. Kihl, “Impact of etcd deployment on Kubernetes, Istio, and application performance”. Softw Pract Exper, 2020, pp. 1-22, https://onlinelibrary.wiley.com/doi/full/10.1002/spe.2885

[30] M. R. Saleh Sedghpour, C. Klein and J. Tordsson, “An Empirical Study of Service Mesh Traffic Management Policies for Microservices”, 13th ACM/SPEC International Conference on Performance Engineering, 2022, https://dl.acm.org/doi/10.1145/3489525.3511686

[31] M. Jøsok Nesset, 2021 “Evaluating Performance And Security Characteristics Of Service Mesh Technologies In A Rancher 2.X Environment”, Bac helor thesis, Norwegian University of Science and Technology (NTNU), Trondheim - https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2781191

[32] Z. Sun, “Latency-aware Optimization of the Existing Service Mesh in Edge Computing Environment”, Dissertation, 2019 - https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1334280&dswid=-3305

[33] N. Dattatreya Nadig, “Testing Resilience of Envoy Service Proxy with Microservices”, Dissertation, 2019 - https://www.diva-portal.org/smash/get/diva2:1372122/FULLTEXT01.pdf

[34] M. O'Keefe, J. Howard, M. Jog, “Best Practices: Benchmarking Service Mesh Performance”, 2019 - https://istio.io/latest/blog/2019/performance-best-practices/

[35] Istio Documentation, “Performance and Scalability” - https://istio.io/latest/docs/ops/deployment/performance-and-scalability/

[36] W. Morgan (Linkerd), “Benchmarking Linkerd and Istio”, 2021 - https://linkerd.io/2021/05/27/linkerd-vs-istio-benchmarks/

[37] W. Morgan (Linkerd), “Linkerd Benchmarks”, 2019 - https://linkerd.io/2019/05/18/linkerd-benchmarks/

[38] R. Li, “Benchmarking Envoy Proxy, HAProxy, and NGINX Performance on Kubernetes”, 2019 - https://www.getambassador.io/resources/envoyproxy-performance-on-k8s/

[39] G. Dillon, “Benchmarking 5 Popular Load Balancers: Nginx, HAProxy, Envoy, Traefik, and ALB”, 2018 - https://www.loggly.com/blog/benchmarking-5-popular-load-balancers-nginx-haproxy-envoy-traefik-and-alb/

[40] Envoy Nighthawk Documentation - https://github.com/envoyproxy/nighthawk/blob/main/README.md

[41] Linux Manual for taskset - https://man7.org/linux/man-pages/man1/taskset.1.html

[42] R. M. Jösch, “Managing Microservices with a Service Mesh”, Degree Project, 2020 - http://kth.diva-portal.org/smash/get/diva2:1464891/FULLTEXT01.pdf

[43] New Relic Docs, “Request queuing and tracking front-end time” - https://docs.newrelic.com/docs/apm/apm-ui-pages/features/request-queuing-tracking-front-end-time/

[44] MDN Contributors, “Understanding latency”, 2022 - https://developer.mozilla.org/en-US/docs/Web/Performance/Understanding_latency

[45] Istio Documentation, “Istio Standard Metrics” - https://istio.io/latest/docs/reference/config/metric