A Modern Monolith Architecture

Aaron O'Brien

Supervisor: Mr. Jerry Horgan

School/Faculty of Science and Computing
South East Technological University
December, 2024

A dissertation in fulfillment of the requirements of the degree of Masters of Science (Enterprise Software Systems).

Contents

1 Introduction **4**
- 1.1 Motivation and Problem Statement . . . 4
- 1.2 Research Questions . . . 4

2 Literature Review **4**
- 2.1 Literature Review Summary . . . 8

3 Methodology **8**

4 Case Study **9**
- 4.1 A Legacy Solution . . . 9
- 4.2 A Modern Monolith . . . 11
- 4.3 Organizational Context . . . 11
- 4.4 Data Notes . . . 11
- 4.5 Architecture Comparison Study . . . 12
- 4.6 Retrieve Item List . . . 13
  - 4.6.1 Microservice Legacy Architecture . . . 13
  - 4.6.2 Microservice Legacy Architecture: Downstream Service . . . 18
  - 4.6.3 Modern Monolith Architecture . . . 20
- 4.7 Item Details . . . 23
  - 4.7.1 Microservice Legacy Architecture: Item Details . . . 24
  - 4.7.2 Modern Monolith Architecture: Item Details . . . 28
  - 4.7.3 Scope Limitation . . . 32
- 4.8 Discussion . . . 32

5 Research Question 3: Common Design Patterns **33**
- 5.1 Containerization and Orchestration . . . 34
- 5.2 Observability . . . 34
- 5.3 Continuous Integration and Deployment . . . 34
- 5.4 Scalability . . . 35
- 5.5 Resilience and Fault Tolerance . . . 35
- 5.6 Security . . . 35

6 Threats To Validity **35**
- 6.1 Selection Bias . . . 35
- 6.2 Extraction Bias . . . 35
- 6.3 Conclusion Bias . . . 36

7 Conclusion **36**
- 7.1 Additional Observations . . . 36

A Appendix A: LOWESS Diagram **37**

List of Figures

1 Microservice Architecture . . . 9
2 Modern Monolith . . . 10
3 Login events per minute . . . 12
4 Magnified view of Login events per minute for first day . . . 13
5 Microservice Plot Trimmed Data Set Sorted by Elapsed Time . . . 15
6 Microservice Scatter Plot Response Size and Response Time . . . 16
7 LOWESS Microservice Diagram Days 1-4 . . . 17
8 LOWESS Microservice Diagram Days 5-7 . . . 17
9 LOWESS Microservice Diagram Days 8-11 . . . 17
10 LOWESS Microservice Diagram Days 12-14 . . . 17
11 Downstream Service Sorted by elapsed time . . . 19
12 LOWESS Diagram Downstream . . . 19
13 LOWESS Diagram Two Day Comparison . . . 20
14 Modern Monolith Sorted by elapsed time . . . 21
15 Monolith Scatter Plot Response Size and Response Time . . . 22
16 LOWESS Monolith Details Days 1-4 . . . 22
17 LOWESS Monolith Details Days 5-7 . . . 23
18 LOWESS Monolith Details Days 8-11 . . . 23
19 LOWESS Monolith — Days 12–14 . . . 23
20 Sorted transactions for item details . . . 25
21 Micro Services Item Details . . . 26
22 LOWESS Diagram Microservice: Item Details Days 1-4 . . . 27
23 LOWESS Diagram Microservice: Item Details Days 5-7 . . . 27
24 LOWESS Diagram Microservice: Item Details Days 8-11 . . . 27
25 LOWESS Diagram Microservice: Item Details Days 12-14 . . . 28
26 Monolith Item Details Sorted by Elapsed Time . . . 29
27 Monolith Item Details Scatter Diagram . . . 30
28 Monolith Item Details LOWESS Diagram Days 1-4 . . . 31
29 Monolith Item Details LOWESS Diagram Days 2-7 . . . 31
30 Monolith Item Details LOWESS Diagram Days 8-11 . . . 31
31 Monolith Item Details LOWESS Diagram Days 12-14 . . . 32

List of Tables

1 Architecture Comparison . . . 8
2 Microservice Metrics . . . 14
3 Downstream Metrics . . . 18
4 Microservice Metrics: Item List . . . 20
5 Microservice Metrics: Item Detail . . . 24
6 Monolith Architecture Metrics: Item Details . . . 28

Abstract - Around 2012, the term microservices emerged. This term is the formalization of a distributed computing architecture pattern. This pattern defines how a single application may leverage multiple services to provide the required functionality. These services are typically small in nature, focusing on a single piece of functionality. In contrast to this new pattern is the monolithic architecture. The monolith architecture pattern typically defines an application that runs as a single process on a server and connects to a single database.

As organizations began to adopt microservices, the complexity of managing these systems became a challenge. This inadvertently led to a new sub-industry created to manage this complexity. New concepts such as orchestration frameworks were created to manage clusters of servers. There were significant gaps in performance and stability monitoring as well as in event logging. These challenges led to the creation of new services and applications to solve these problems. Implementing an orchestration, centralized monitoring, and logging adds significant complexity to robust microservice deployment. Ultimately, this complexity leads to an increase in the cost and time required to deliver an application. This impact can be avoided by beginning with a simpler architecture. There is mounting evidence showing that, although a microservice architecture improves resilience and stability, it also negatively impacts performance, speed to delivery and overall cost. The goal of this paper is to take the lessons from the microservice architecture and use them to define a new architecture pattern called the modern monolith. The modern monolith pattern creates a straightforward, understandable code base that is resilient and able to scale based on the demands of real-time traffic patterns and performance goals.

There is no best architecture pattern that works for every application. However, the adoption of the microservice pattern prematurely will provide the expected benefit when the application meets certain criteria. This study examines these architectures and metrics to discover the architectural solution that provides the highest return on investment for a project.

1 Introduction

The microservices pattern was formalized around 2012 [1] and since then, companies like Netflix, Amazon, Uber, and many others have been able to achieve high levels of success with reliability, scalability, and performance by converting their existing monolith applications to microservices. Based on the results of these use cases, it is clear that microservices are foundational to the success of these companies. The success of these companies has led many other organizations to attempt to follow along without understanding the impact that microservices will have on their speed of delivery, budget, and organizational structures.

In this study, when referencing a monolithic architecture, I am referring to a design pattern called a **modern monolith**. Similar to a traditional monolith, a modern monolith can be run as a single process that connects to a single database. However, with the maturation of technologies like containerization and orchestration frameworks the modern monolith is able to take advantage of these tools to improve on the original monolith design. Using modern tooling in this way creates a consistent, high-performing, highly scalable, resilient solution that has a smaller attack surface area and is more easily managed and deployed. This study will compare a microservice architecture with a modern monolith architecture, gathering real-world metrics taken from both systems that are running in parallel in a production-level environment.

1.1 Motivation and Problem Statement

The problem that I am attempting to address with this study is that in a modern computing business environment there is a trend to build microservices first. With the belief that building independent parts (or services) that can be assembled as needed provides a faster, cheaper, more resilient application.

I have found this early adoption of microservices to be an anti-pattern and this study will attempt to show that a modern monolith has many of the advantages of a microservice and can also be delivered even faster achieving better overall performance in the early stages of an application's life cycle.

1.2 Research Questions

There are many publications that compare the monolith and microservice architectures. Many have similar points around focusing on the resilience and scalability of microservices. However, these papers have a common trend in that they are showing a window where monoliths perform better than microservices. A post by Martin Fowler [2] talks in depth about a practical approach based on two key points:

> *Almost all successful microservice stories have started with a monolith that got too big and was broken up.*
>
> *Almost all stories of a system built with a microservices first approach end up in serious trouble.*

According to the research, it appears that monolith architectures should be the first step in building a new application. In the original monolithic architecture the performance will eventually degrade as the application takes on more load. The questions this paper will attempt to answer are:

- By modifying the definition of a monolith, are there lessons from the microservice architecture that could improve the scalability and resilience of a monolith?
- What are the key metrics that would indicate that a modern monolith architecture should begin to move towards a microservice architecture?
- Are there common design patterns that can be leveraged by both architectures, and how does the architecture change the implementation of these patterns?

2 Literature Review

The goal of this study is to understand the pros and cons of the microservice and modern monolith architectures. To understand these pros and cons there are several papers that were used to formulate the foundation of this research.

Around 2012, the microservice architecture pattern was identified, and due to the benefits such as improved availability, fault tolerance, horizontal scaling and greater software development agility, companies took notice and started moving their existing systems to take advantage of these benefits. In the paper "Monolithic vs Microservice Architecture: A Performance and Scalability Evaluation" [1] the authors assert that systems that do not have thousands of concurrent users, scaling resources vertically (i.e. increasing server resources such as CPU, memory, etc.) will allow higher levels of performance. There are two main questions that this paper answers that help inform this research:

- Question 1: What is the performance difference between monolithic and microservice applications? Based on the research, the authors concluded that a monolith performs better. This is largely due to the increased overhead of passing requests from one microservice to another.
- Question 2: Which of the two architectures and scaling approaches should be chosen to best benefit an application? By leveraging the ability to scale an application vertically, the monolith was able to achieve higher levels of performance. Vertical scaling increased the performance of microservices as well. However, there was a cap on the number of instances a microservice ran on and increasing the number of instances over that amount did not provide an improvement in performance and may degrade performance. Over-scaling manifests when CPU overhead results from load distribution - in other words, it can add time to choose the correct instance to route traffic to if there are too many instances to choose from.

This paper noted that in many cases, a monolithic application can perform better than a microservice application. However, there is a threshold of concurrent users where the performance of a monolithic architecture will begin to degrade. This paper did not attempt to define this threshold, although it did mention that unless an application handles several thousand concurrent requests, the distributed architecture of microservices will likely increase cost and complexity unnecessarily. Many companies will make the decision to convert a monolithic application to a microservice. Prior to making that decision, the paper "Migration from Monolith to Microservices: Benchmarking a Case Study" [3] discusses a methodology that could be used to verify if the migration will have the expected benefits. Before going into the actual work of the study, an interesting observation was made that referenced *Conway's law* - which loosely states that the systems a company builds - are a reflection of the communication and organizational structure of the company. This is an important observation, because a company that culturally engages with many different stakeholders to make decisions, it is unlikely that the independent nature and agility that microservices provide will fit in with the structure that this company would have. Microservices will align better with a company that has smaller teams that are independent and empowered to make their own decisions. This paper went into detail on different types of measurements that have been used to analyze different applications. Although many different metrics were identified, the following metrics were used as the most important in this study: latency, throughput, scalability, memory and network. These are important concepts for every modern application. *Latency* is the amount of time it takes to get a response from a server after making a request; there are many variables that affect this measurement: the amount of processing it takes, but also the number of network calls that are made affect timings around latency. *Throughput* is the amount of requests a service can handle simultaneously; similar to latency, there are many things that affect this metric - for example, the CPU demand of each request affects this metric greatly. *Scalability* is talked about many times in almost every paper, although there are two basic types of scalability (horizontal and vertical); both of these types of scaling are basically adding more capacity to the overall system. As discussed earlier vertical scaling is adding more resources such as CPU, memory, storage, etc., while scaling horizontally adds new instances of servers to share the load of the current demand. *Memory* is the amount of RAM memory available to a server. The Network metric comes down to how many requests are made with in the network; for example - a monolithic has a single request and response cycle to the client, whereas a microservice application has a much higher network usage as each request flows through a gateway service and any number of other microservices to fulfill the needs of the request. A new application was created in this study and implemented in two different architectures to assess these different metrics. In almost all scenarios that were tested the monolith application performed better than the microservices. Instead of testing these systems under a high load, the authors algorithmically derived the performance of how a system under scale would perform. Based on the article by Blinowski [1], where over-scaling can occur it seems that an algorithmic approach to performance is not a good baseline for this assessment. However, the conclusion of this paper was in line with Blinowski's research stating that applications with a small to medium size load would benefit more from a monolithic architecture than a microservice application. The metrics

chosen by Nicholas Bjorndal in the paper "Migration from Monolith to Microservices: Benchmarking a Case Study" [3], that was just reviewed, focused on latency, throughput, scalability, memory and network. A different paper by Milic [4] focused on different metrics. That paper focused on coupling, testability, security, complexity, deployability and availability. The paper uses those metrics to assess which architecture provided the highest levels of quality. This paper makes an interesting observation that the most important step in software development is the software design and the selection of the software architecture is used to describe the software design and represent the basis for further software development processes.

Milic defines these key metrics as follows:

- Coupling is the interdependence of modules of software.
- Testability covers anything from unit tests to end-to-end tests.
- Security examines how the data is protected; access controls are important here as well.
- Complexity is defined as how difficult a component is to be understood and verified.
- Deployability assesses the ease in how different environments can be leveraged (development, staging, production, etc) as well as the artifacts themselves.
- Availability measures the degree to which a system is functional.

These metrics lead to a very different perspective on how to assess the different software architectures. To do this assessment, the team created two different versions of the same application and assessed each version on these metrics. *Coupling* showed a clear advantage to microservices. Monolith systems regardless of their design will have inter module dependencies. Although dependencies still exist in mircorservices they are separated in a way that allows for individual microservices to be changed with less of an impact to the overall application. The testability in respect to unit tests, seemed to be more straightforward with monoliths. In regards to *testing*, the goal of this study was to achieve 100% unit test coverage and the monolith application was able to achieve this goal. However due to the high level of aspect-oriented programming there were areas where unit testing could not be created. This resulted in a final test coverage of 86.3%.

The security assessment of the two applications showed that the security concerns that a single monolith has are the same concerns that each individual microservice has as well, on top of those concerns each microservice also needs to secure the communication channels between them. This assessment greatly increases the surface area that needs to be secured in a microservice architecture. Assessing *complexity* is about how difficult is the code base to understand. This is a more difficult thing to understand when working with the larger codebase of a monolith. In this assessment microservices are able to break down complexity of the overall system allowing development teams to focus on a single part of the overall application.

In the review of *deployability* several things were looked at compile time, size of the deployment, and support for different environments. The compile time for a single monolith was approximately 2 minutes where different microservices were around a minute and a half. Similarly, the size of the monolith was 41.2 MB but the different microservices ranged from 40.4MB to 53.2MB. This led to the observation that microservices introduce a lot of duplicated code into a system. The support for different environments was equal between the systems. The assessment of availability clearly showed that microservices improved the availability of the overall system.

The overall conclusion was that there is a good case for both monolith and microservice architectures. Having strong metrics for evaluating the attributes a company has the most concerns over should be paramount to deciding which architecture is better. If a company is primarily concerned about one or two attributes, they should pick the appropriate architecture in alignment with their needs; however there are mitigating solutions that can offset disadvantages to any particular architecture. For example security seems to be better managed with a monolith, but with extra time and effort the security challenges with microservices can be overcome. By understanding the driving concerns of a company, these metrics will help the company make those decisions better. The previous article by Blinowski [1] went into an in-depth discussion about metrics that measure systems from a quality perspective the paper titled "Performance Evaluation of Monolithic and Microservice Architecture of an E-Commerce Startup" authored by Nahid Nawal Nayim [5] evaluates the contrasting performance of the two architectures. This paper suggests that an application designed under the monolithic architecture can have the potential to

decrease infrastructure expenses and enhance performance when compared to the microservice version. This paper goes into a discussion about the goal of software architecture to establish a more effective separation of concerns. There are many solutions to accomplish this goal and the software design must reflect the architecture to effectively achieve this goal. There are many variables that go into choosing an architecture and this decision is important as the architecture has an impact on the performance, scalability and cost-effectiveness.

Similar to other papers reviewed the monolith architecture is defined as an application that runs in a single process on the host computer. This is an interesting statement because with modern applications (Javascript, React, Angular, etc.) the user interface is downloaded and run on the user's computer (browser) while the service is running in a single process to handle all requests from the user interface. With microservices, the user interface follows the same pattern although all requests typically go through a gateway service which is then distributed to the appropriate microservice and then transferred back to the client.

Cost is a key metric in this paper. In the application that was implemented, the Monolith had a single EC2 instance and a single DB instance that was allocated the final monthly cost was $310.98. The Microservices had 4 EC2 Instances at monthly costs of $127.02 each, along with a Database, ElastiCache, Amazon MQ and a load balancer this ended up with a final monthly cost of $790.30.

Although there are some optimizations that could be done to optimize the costs, but there is a valid point: it would not be unusual for a small to medium size application to have 10-20 different microservices.

Each unique microservice will increase the cost of the cloud architecture. As the application was configured in this paper with a virtual user count of 50,000 users the monolith architecture was able to perform 89.64 requests per second with the microservice serving 76.57 requests per second. That is a significant difference on a per second basis. The majority of the delay were the requests being processed through the gateway application, Rabbit MQ, and the Redis Cache. Although these extra components did add some latency in the request if scaling software is the primary concern these are good trade offs because as the user base grows the monolith architecture will eventually find its limits. The challenge is defining what that limit is. In the first paper reviewed [1], it seemed that limit could be somewhere around 20,000 users and this paper used 50,000 users. The actual number will likely be unique to the individual system depending on how CPU intensive each request is. In all of the papers thus far, few have done more than mentioned security as something that should be considered. The paper "Fortifying Information Security:Security Implications of Microservice and Monolithic Architectures" by Sanghamitra Nemmini [6] focuses on this subject. This paper starts with acknowledging the limitations of a Monolithic Architecture and states that the microservice architecture seems to solve the majority of those concerns. However, there are several security concerns - specifically for web based applications that need to be addressed. Software Architecture plays a large role in the security of an application. The popularity of the microservice architecture has been growing rapidly since 2016. When comparing the two architectures, it is clear that the simpler nature of the monolith reduces the surface area of attack. The main concern really is at the entry points of the application. A microservice architecture has multiple attack surfaces. Each microservice represents a point of possible attack. This can be a difficult challenge to overcome, but there are benefits in this complexity. For example, if one microservice is being attacked it can be taken offline and provide the potential for the remaining application to remain functional. There are several key points this paper makes that make great points. Communication, debugging and testing can be complex as we've seen in other papers. Having a staff of experienced engineers that understand this complexity is important to the overall success, in comparison, a team of less senior engineers could successfully implement a monolith and secure it successfully. Monitoring is essential in every microservice. Understanding this status and knowing in real time when an attack is happening so that steps can be taken to isolate a service will help significantly. Authentication and Authorization should happen at every service, often it is assumed that if a service is on a secure network, it is safe. This has been proven many times to not be true.

In conclusion, this paper states again that each architecture has its strengths and weaknesses. If an application is smaller in size with a smaller user base, the difficulty to secure a microservice architecture may not be the right choice. Both systems can be considered secure. It is worth noting that microservices have the added benefit of being resilient to attacks where the application performance may be degraded, but still stays functional.

2.1 Literature Review Summary

This literature review 8 different papers looking at the advantages and disadvantages of microservices and monolithic architectures. The authors of these papers shared a deep analysis from a variety of perspectives. Based on these papers, table 1 shows a high-level summary of how a monolith, a modern monolith and a microservice compare with one another.

Metric	Classic Monolith	Modern Monolith	Microservice
Complexity	low	low	high
Implementation Velocity	high	high	low
Performance (small user base)	high	high	medium
Performance (large user base)	low	high	medium
Scalability	low	medium	high
Resilience	low	medium	high
Security	high	high	low

Table 1: Architecture Comparison

One of the interesting observations from the literature review which was repeated in many of the papers, is that when the user base is low (below 10-20K) a monolith will typically perform better than a microservice. The exact number can not be defined as it is dependent on how many resources(CPU, memory, etc.) are needed to fulfill the request as well as the hardware configuration. Monitoring performance of the overall system is key to understanding when the threshold is being approached.

There are many different ways to define what is the best architecture. If a company is highly focused on security, a modern monolith may be chosen for the simplicity of securing it. If scalability and resilience are the most important aspects then microservices may be the best choice. This is not to say that these limitations cannot be overcome, but different application architectures have advantages over others.

3 Methodology

A case-study approach will be used to apply this research using a modernization project that began in 2024. This project is modernizing an existing system that has been built with a microservice architecture and is being simplified to consolidate these systems into a modern monolithic architecture.

The research for this paper will be executed by gathering data from the different architectures and comparing the performance of both architectures. Data will be gathered while the applications are running under normal performance loads. By accessing services that provide similar functionality in both systems, a performance baseline between the systems will be established. Metrics will be gathered on these systems every minute over the course of at least 14 days. This will generate statistically relevant data while the system is under load and while it is idle. These metrics will provide a measure of the elapsed time of each request and the amount of data that is returned. This data should provide indications of how the systems behave in different scenarios, such as high and low volume usage throughout the period of time. This data may help point to other problems that would lead to gathering more metrics related to dependent systems or resource utilization constraints.

The timing of this research is in a period of time where both the legacy microservice and the new monolithic architecture are in production. Users are actively being migrated from the legacy system to the new system. This will provide a window where metrics can be gathered on both systems at the same time, allowing the analysis of performance data when the systems are under load and when they are idle. The plan is to look at concurrent users who are using the existing system and how the load they create affects metrics such as latency and throughput in the system.

These metrics will become the sources of evidence to further the research on this subject and help draw conclusions for the research questions that I documented earlier in this paper. For convenience, I have listed the questions below:

- *By modifying the definition of a monolith, are there lessons from the microservice architecture that could improve the scalability and resilience of a monolith?*
- *What are the key metrics that would indicate that a modern monolith architecture should begin to move towards a microservice architecture?*

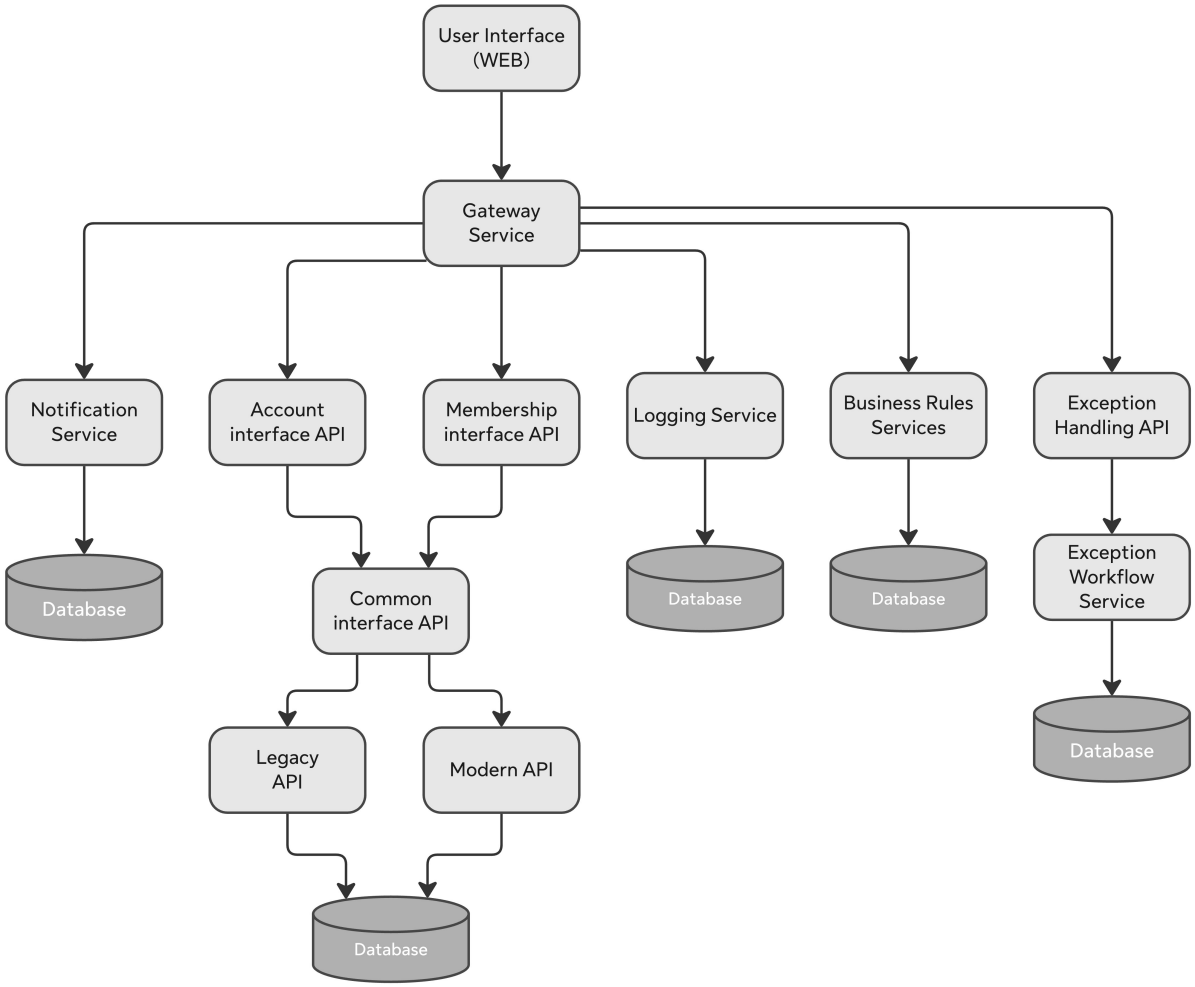

Figure 1: Microservice Architecture

- *Are there common design patterns that can be leveraged by both architectures, and how does the architecture change their implementation?*

4 Case Study

This research paper analyzes a system that has been in production for many years. This system was written using a microservice architecture. A new project was started in 2024 to redesign the application using a modern monolithic architecture. Currently, both systems are running side-by-side, and there is a unique window of opportunity to compare both systems to gain a better understanding of the advantages of the different architectures and determine which metrics to monitor that would indicate that the new architecture should be moving towards a microservice architecture.

4.1 A Legacy Solution

The legacy system is a combination of more than fifty different microservices working together to provide a final solution for the customer. A simplified version of this architecture is represented in Fig. 1. The application follows a fairly standard pattern of a user interface talking to a single service that becomes a gateway for a variety of other services. These other services range from sending asynchronous notifications, different APIs to different source systems that manage account and member details, event logging, a variety of different business rules for different use cases, and other services.

These services were written prior to the adoption of containerization technologies, and each service is running as a pair of services that are load balanced. Although less resilient than an orchestration framework, this configuration has performed well over time.

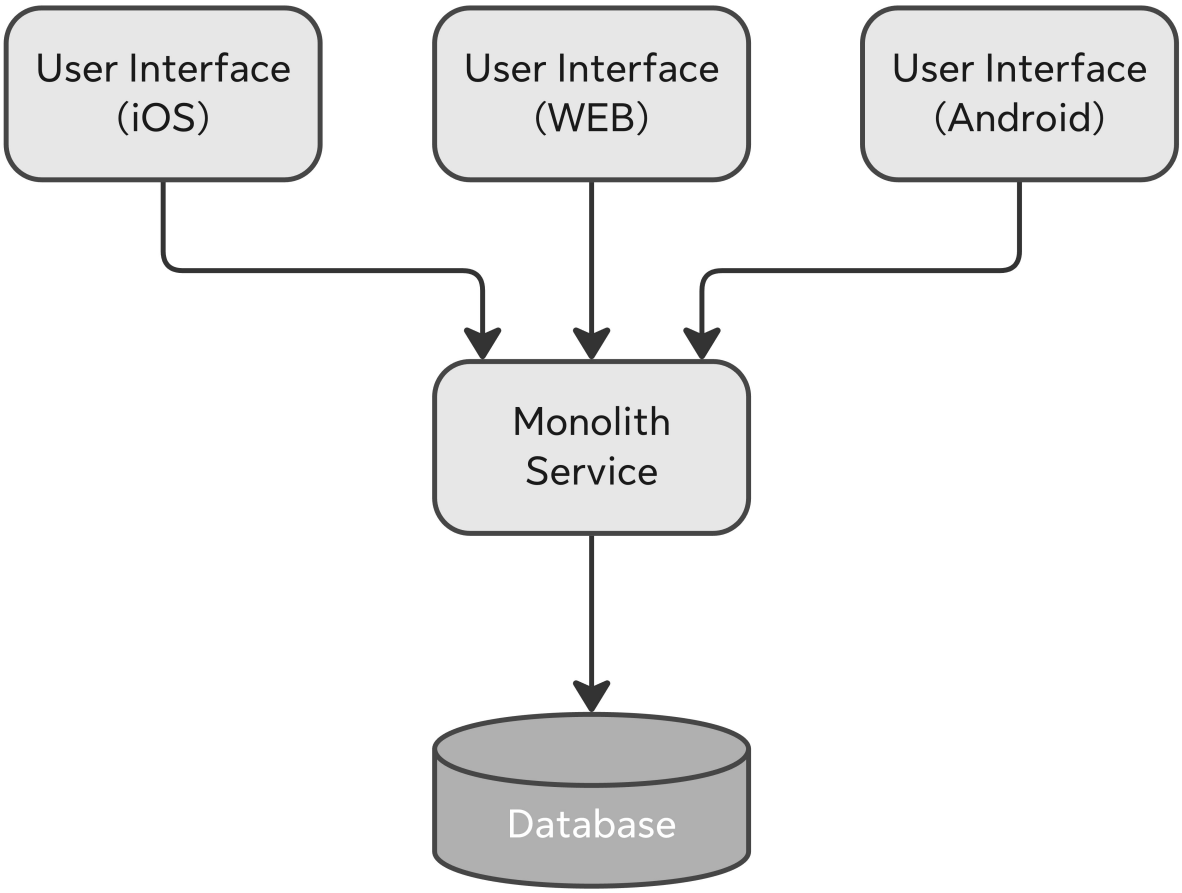

Figure 2: Modern Monolith

4.2 A Modern Monolith

The first research question for this study was defined as:

> *By modifying the definition of a monolith, are there lessons from the microservice architecture that could improve the scalability and resilience of a monolith?*

From the literature review, a monolith is defined as a system that runs on a single server in a single process. If that definition is adapted and the capabilities of containerization and an orchestration framework are introduced, the end result is a self-contained application that can be auto-scaled and have resilience integrated through the orchestration framework.

A monolith application typically maintains the user's state and context as the application is being used. This concept of state and context management must be redesigned to successfully implement a modern monolith architecture. If a containerized service attempts to maintain state, such as items in a shopping cart, there is no guarantee that the next request from that user will go to the same container, potentially causing errors and unexpected behavior. To properly build a containerized application, user state cannot be held in the memory.

This methodology is the basis for a modern monolith that performs well with all levels of user traffic, has a manageable code base, and is still scalable and resilient. This architecture is depicted in Fig. 2.

The new version of the system that is being analyzed is built as a modern monolith as described. The modern monolith consolidates the functionality of the legacy microservice layer into a unified application and data model. This modern monolith is also built by leveraging modern technologies such as React for the user interface and GoLang for the service layer.

Having a consolidated data layer reduces the need to call APIs to these other systems. In modern software development, APIs have become the standard way of communicating between systems. There are many advantages to using this distributed pattern, and we will investigate some of the advantages and trade-offs with both patterns.

Fig. 2 also depicts several potential user interfaces that could be implemented to take advantage of the monolith service. This monolith is responsible for sending asynchronous notifications, accessing the consolidated data model supporting account and member details, event logging, a variety of different business rules for different use cases, and many other functional areas.

4.3 Organizational Context

The organization that has developed these applications is a multi-national corporation specializing in selling insurance to other companies that want to offer their employees insurance to provide financial support in the event of an unexpected event in their life. The user base of this application is an employee of the company receiving benefits, and they have been designated to manage the employees of that company in this application. Managing employees in this sense is defined as viewing members as well as validating selected benefits and dependents. There are a variety of insurance products that a user could be offered to enroll in. This study will review the APIs that allow user to access a list of employees and the details for a specific employee.

Permission was granted in this organization to perform this study and set up a metrics gathering system that was able to measure the systems within a subset of the organization. The monolith was able to be fully tested. Unfortunately, accessing the different microservices that were maintained by a number of different teams proved to be a difficult challenge that created limitations in my analysis. For example in the initial analysis of the microservice it was decided to retrieve the logs from the downstream service for that same period of time and only a limited set of data was available. It appeared that the bottleneck was even further in the architecture and accessing performance data of the database was not available to the study.

4.4 Data Notes

Five data sets were retrieved for this study:

1. Retrieve a paginated list of employees accessing the microservice architecture
2. Retrieve the events for the downstream service that correspond to the microservice list of employees
3. Retrieve a paginated list of employees accessing the monolith architecture

4. Retrieve the detailed information for an employee accessing the microservice architecture

5. Retrieve the detailed information for an employee accessing the monolith architecture

Although in normal operations these services have the potential to access sensitive data. The system that was built to gather metrics never accessed the content of the responses, only the metadata around the request was recorded (status of the response, size of the response and how long it took to process the request).

For the first set of data in the list above data was gathered from Saturday, April 19th at 00:00.00 to Friday, May 2nd at 23:59.59 this was a full 14-day cycle.

For the second set of data in the list above the data was retrieved from a separate system that had limitations on how much data could be retrieved; only five days were retrieved and this range of data was from Monday, April 28th to Friday, May 2nd.

For the third set of data in the list above, the data was gathered from Saturday, April 29th at 00:00:00 to Friday, May 12th at 23:59:59. This was a full 14-day cycle.

For the fourth set of data in the list above, the data was gathered from Tuesday, June 24th to Monday, July 7th.

For the fifth set of data in the list above, the data was gathered from Friday, June 18th to Thursday, July 1st.

There was no significance to the day of the week, month or time of the month when the data was gathered for analysis. The goal was to collect 14 days of continuous readings to capture the same number of days for each category of data. The intent was that if there was a significant change in behavior over the weekend or on specific days of the week, having a record of multiple days should expose those trends.

4.5 Architecture Comparison Study

There is a period of time where the legacy microservice and the modern monolith will be running in a production environment at the same time. This window will provide an opportunity to validate the overall performance of both systems while under a production load.

A metrics-gathering system was created to record the performance of both systems. Metrics were gathered every minute for approximately two weeks, providing approximately 20,000 data points for each test and offering a view of system performance while under load and while mostly idle.

The process of gathering metrics was to make an API call and to record the following data points:

- Request Start: This was measured as a timestamp to the millisecond
- Request End: This was measured as a timestamp to the millisecond
- Response Size: This was measured in bytes
- Status Code: The HTTP response code was recorded from the API call

In these applications, when a user logs in to access the system, this event is recorded with a timestamp. By extracting and graphing this data (Fig. 3), we can see an overall trend, with Fig. 4 showing a detailed view of a single day. These diagrams show that users begin logging in around 08:00 and consistently log in until around 18:00. This data does not represent user activity or session length. It is illustrating a minute by minute graph on how many people are logging in to the application. The key information from this diagram is that there are over 6,000 logins per day, and the vast majority happen between 08:00 and 18:00 hours.

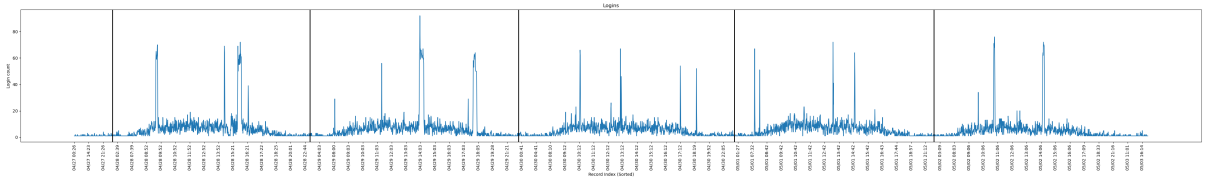

Figure 3: Login events per minute

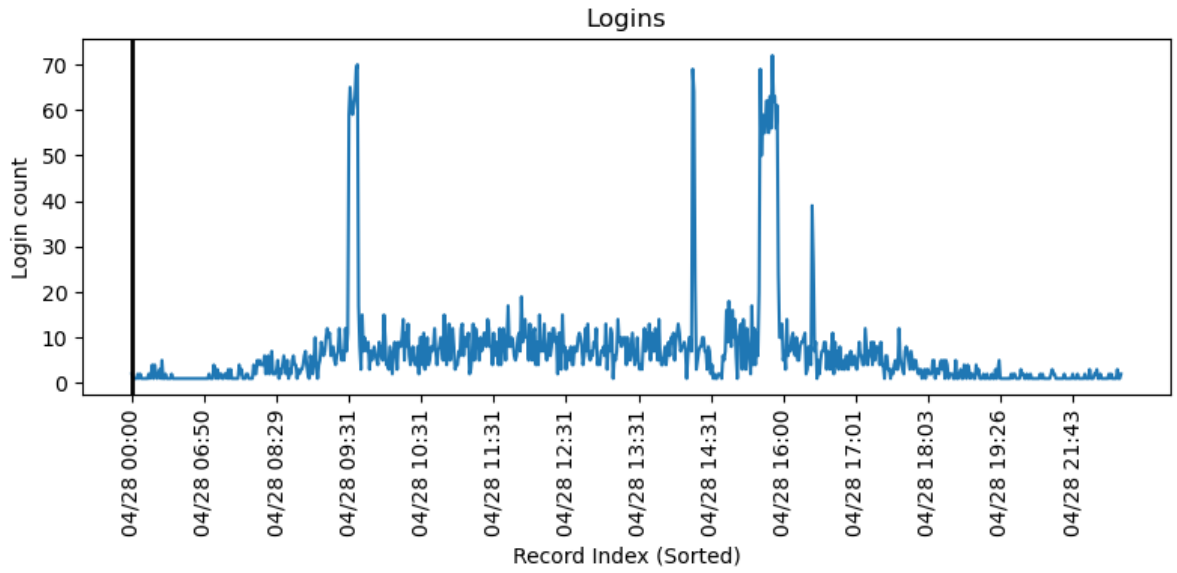

Figure 4: Magnified view of Login events per minute for first day

4.6 Retrieve Item List

The first API call that was measured makes a request to retrieve the data for 20 employees from a company that has over 1,000 employees. The next time a call was made, the next set of 20 employees was requested. Iterating over the full employee list was done to reduce any caching the system(s) might implement, which could skew our test.

The first API call that was measured makes a request to retrieve the data for 20 employees for a company that has over 1,000 employees, the next time a call was made the next set of 20 employees were requested. Iterating over the list of the full employee list was a done to reduce any caching the system(s) may implement that could skew our test. With these data elements being recorded over an extended period of time, data discussed in section 4.5 was generated to assess the overall performance of the system during peak and idle times. A similar call was made on both architectures at different times to prevent any false skewing of performance metrics.

4.6.1 Microservice Legacy Architecture

The microservice architecture was tested for 14 days, generating over 20K measurements that covered peak times, idle times and weekend times. The breakdown of this data is shown in table 2. This table shows metrics for the complete data set and the data set after being trimmed. By trimming the fastest 0.5% and slowest 0.5% of transactions, we remove one percent of the data, or 284 transactions. In a live production system, there are many reasons transactions could be extremely fast or extremely slow. Factors such as a long-running garbage collection cycle or a networking error could increase call durations significantly. Similarly, load balancing could cause data to be cached, allowing calls to be occasionally faster. By removing a half of a percent of the fastest and slowest measurements we can focus more on the statistically relevant data points.

In the data visualizations for this study, unless otherwise specified the trimmed data set will be used.

In scenarios where the standard deviation (STD) is larger than the average (mean) of the data, a situation occurs that is described as a "long tail of latency." This particular pattern describes a situation where a small percentage of requests take disproportionately much longer to complete than the vast majority of requests. This long tail can be visualized by sorting the data by elapsed time of the requests and graphing it using a plot diagram see Fig. 5.

An additional metric to calculate is the Coefficient of Variation (CV), which is a standardized measure of dispersion of a probability distribution or frequency distribution see equation 1. It is defined as the ratio of the standard deviation to the mean.

The formula is given by:

Metric	Complete Data Set	Trimmed Data Set
Mean	442 ms	436 ms
STD	625 ms	185 ms
Min	133 ms	188 ms
Max	48,080 ms	1,425 ms
P25	314 ms	315 ms
P50 (Median)	390 ms	392 ms
P75	485 ms	496 ms
P90	623 ms	654 ms
P95	775 ms	816 ms
P99	1,159 ms	1,148 ms
Data Points	20,149	19,948

Table 2: Microservice Metrics

$$CV = \frac{\sigma}{\mu} \times 100\% \tag{1}$$

where:

- σ represents the standard deviation.
- μ represents the mean.

In this set of metrics if we were to use the complete data set our CV would look like this:

$$CV = \frac{625}{442} \times 100\% = 141\% \tag{2}$$

In the trimmed data set we show that the CV falls into a much healthier region:

$$CV = \frac{185}{436} \times 100\% = 42\% \tag{3}$$

In general terms values for a "good" CV for API Response Times are as follows:

- A low CV (e.g., <10–20%) is generally desirable for critical, user-facing APIs where consistent and predictable performance is paramount. This indicates that most response times are clustered tightly around the average.
- A moderate CV (e.g., 20–50%) might be acceptable for less critical APIs, or during periods of unusual load, but it suggests there's room for improvement in consistency.
- A high CV (e.g., >50%) often points to significant inconsistencies or bottlenecks. This means there's a wide spread in response times, and a good portion of users are likely experiencing much slower responses than the average.

The CV calculated in equation 2 equals 141%, and the trimmed data set in equation 3 shows a CV of 42%. Focusing on the trimmed data set, a CV in this range could be acceptable in non-critical systems.

The CV is just one single measurement, and breaking the data into percentiles (as in Fig. 5) shows that half (50% percentile) of the responses for this API are more than 390 milliseconds.

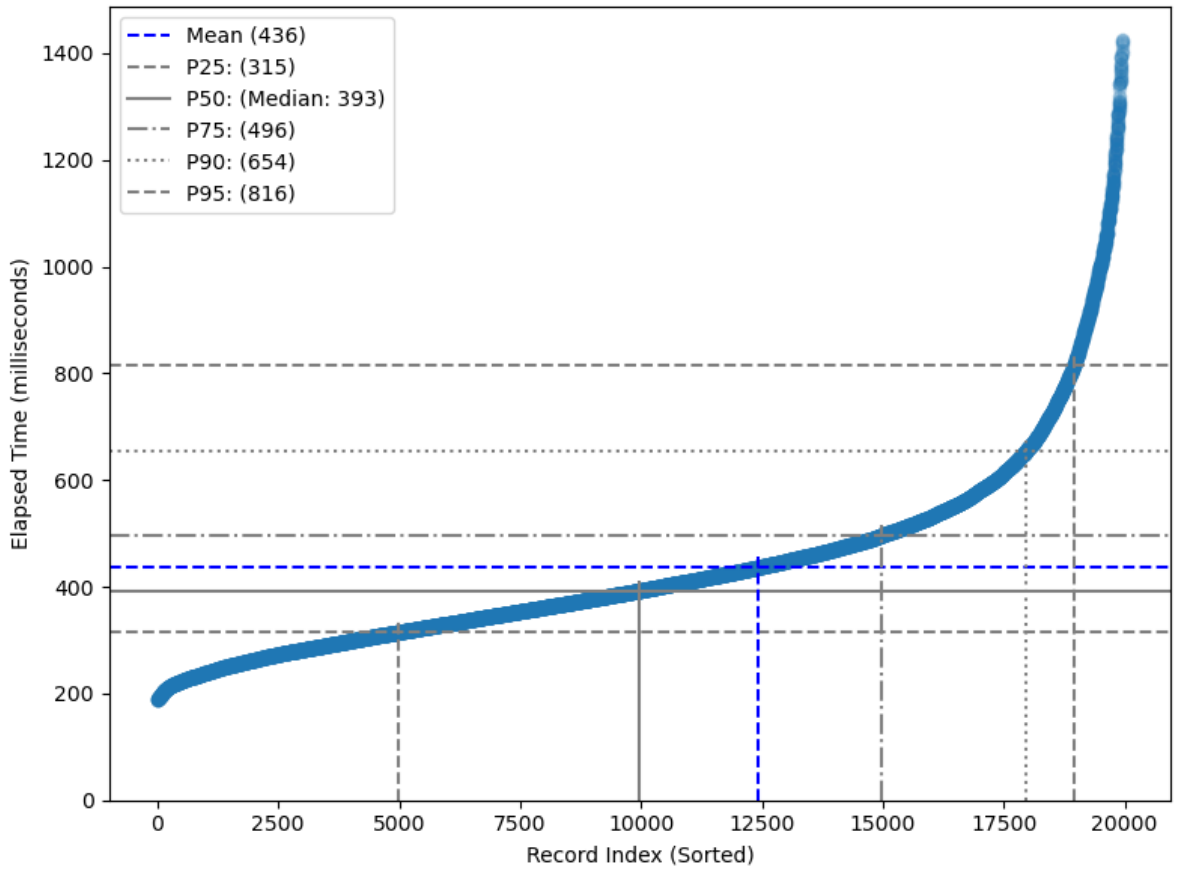

Figure 5: Microservice Plot Trimmed Data Set Sorted by Elapsed Time

Another visualization of this data can be seen by plotting the data into a scatter diagram see Fig. 6. What this diagram is showing is a relationship between the bytes that are returned to the speed of transaction. This diagram shows that the vast majority of transactions are faster than 600 milliseconds. Looking at table 2 we see that 600 milliseconds is somewhere around the 80th percentile. This diagram also shows that the largest transactions have some of the fastest response times. This is showing that for this data set the size of the data is not a factor in the overall performance of the transactions. A scatter diagram is typically used to show a relationship between two different variables. In most networking situations there is a strong relationship between the size of the response and the length to retrieve the response. This is typically is visualized by a diagonal line showing the relationship between these variables.

The scatter diagram in Fig. 6 shows that there is not a strong relationship between the size of the response and the time it took to retrieve the response. Some of the largest responses achieved very fast response as well as many of the smallest responses took very long times.

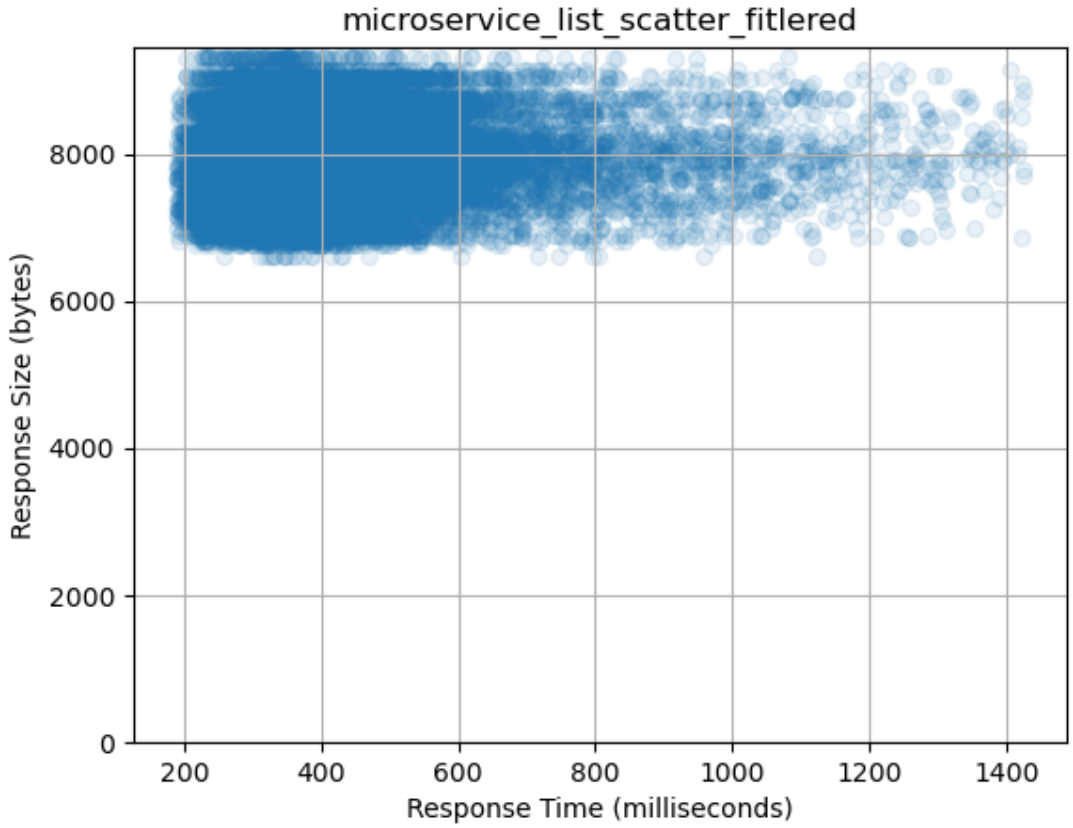

Figure 6: Microservice Scatter Plot Response Size and Response Time

The final diagram used to represent this data is called a LOWESS diagram (see appendix A for a definition). This diagram shows the data in sequence, and to view the details more clearly, the data is broken into days—graphing the first four and the next three days to provide a one-week view (see figures 7 and 8). Fig. 9 and Fig. 10 represent the second week of data.

In these diagrams, every data point measured in the set is graphed as a blue point. Connecting these points produces a thin line that illustrates the raw performance of each measurement. The LOWESS smoothed line uses an algorithm to analyze neighboring data points and plot points that, when connected, provide a more general view of the data. This smoothed line makes it easier to identify performance peaks and valleys.

When putting this graph in the context of the user logins observed in Fig. 3, we see that the best performance occurs during peak usage, while performance appears to degrade every day around 1:00 AM.

This indicates that the bottlenecks in this service are unrelated to user load, memory usage, CPU usage, or other common factors. Instead, there is an issue further down in the architecture that is causing these performance spikes.

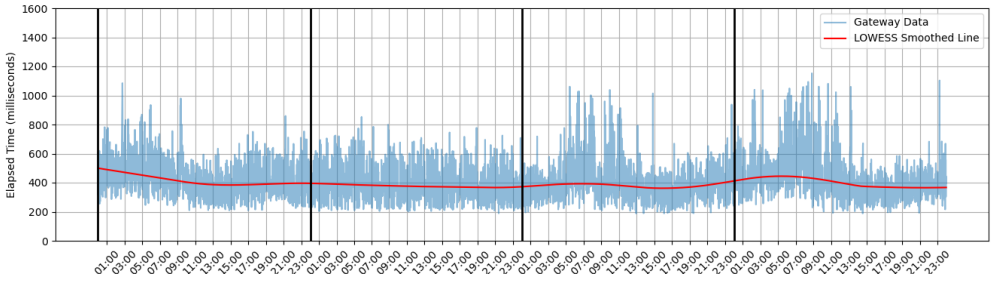

Figure 7: LOWESS Microservice Diagram Days 1-4

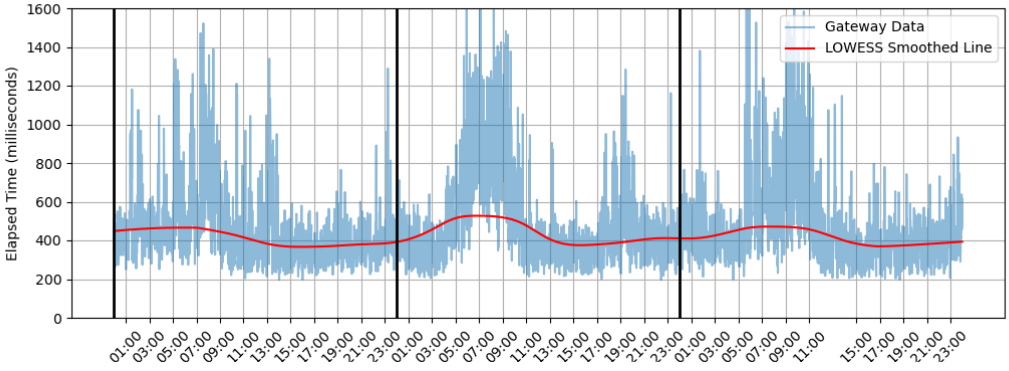

Figure 8: LOWESS Microservice Diagram Days 5-7

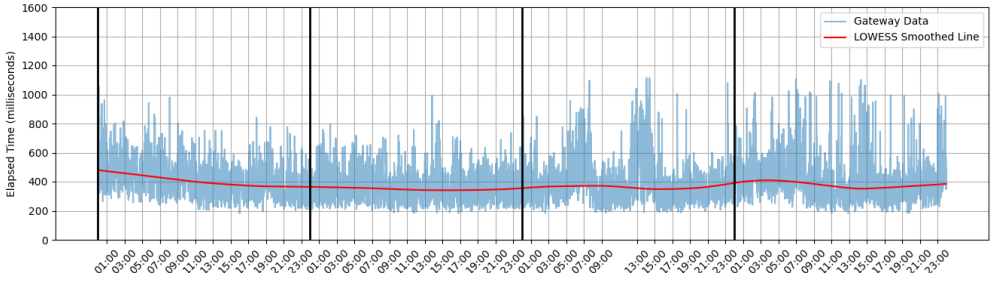

Figure 9: LOWESS Microservice Diagram Days 8-11

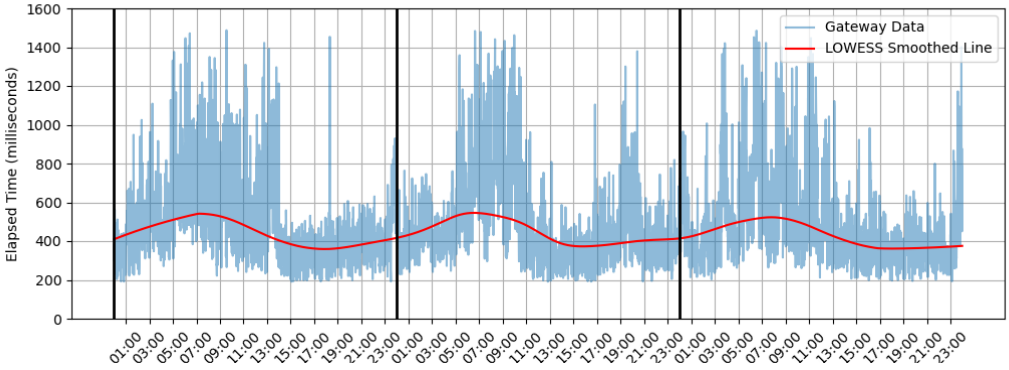

Figure 10: LOWESS Microservice Diagram Days 12-14

4.6.2 Microservice Legacy Architecture: Downstream Service

The performance of the microservice was unexpected. The high levels of variation in calls, as well as the slower performance outside of peak usage hours, indicate that there are unknown variables affecting the performance of the gateway service. Reviewing Fig. 1, the previous metrics we reviewed were focused on the gateway service. This next section goes a level deeper to access a service that interacts with the source system for this data.

To access these metrics, a data request was made and five days of logs were provided from the interface service. These interface services are used by many applications, so there was much more data provided than what was applicable to our test. A series of filters were put in place to find the exact transactions based on start and end times of transactions as well as sizes of responses. Eventually, a 1-to-1 match was found, and the following analysis is provided for the five days of data that were provided.

The data for the downstream system was extracted during the same period of time for five days. The data is shown in table 3.

Metric	**Complete Data Set**	**Trimmed Data Set**
Mean	348 ms	340 ms
STD	422 ms	181 ms
P25	219 ms	220 ms
P50 (Median)	295 ms	295 ms
P75	398 ms	396 ms
P90	568 ms	559 ms
P95	744 ms	727 ms
P99	1,133 ms	1,031 ms
Data Points	7200	7132

Table 3: Downstream Metrics

Calculating the CV using equation 1, we get the following calculation: in this set of metrics we would have:

The CV for the complete data set is calculated as follows:

$$CV = \frac{422}{348} \times 100\% = 121\% \tag{4}$$

While the trimmed Data set is calculated as follows:

$$CV = \frac{181}{340} \times 100\% = 53\% \tag{5}$$

This tells us that the mean is a poor metric to evaluate the overall performance, and this service can be unpredictable.

Similar metrics were gathered, reflecting similar patterns of the "Long Latency Tail," which is identifiable by the STD value (422 milliseconds) being higher than the mean (348 milliseconds) (see Fig. 11). This diagram is almost identical to the gateway service in figure5.

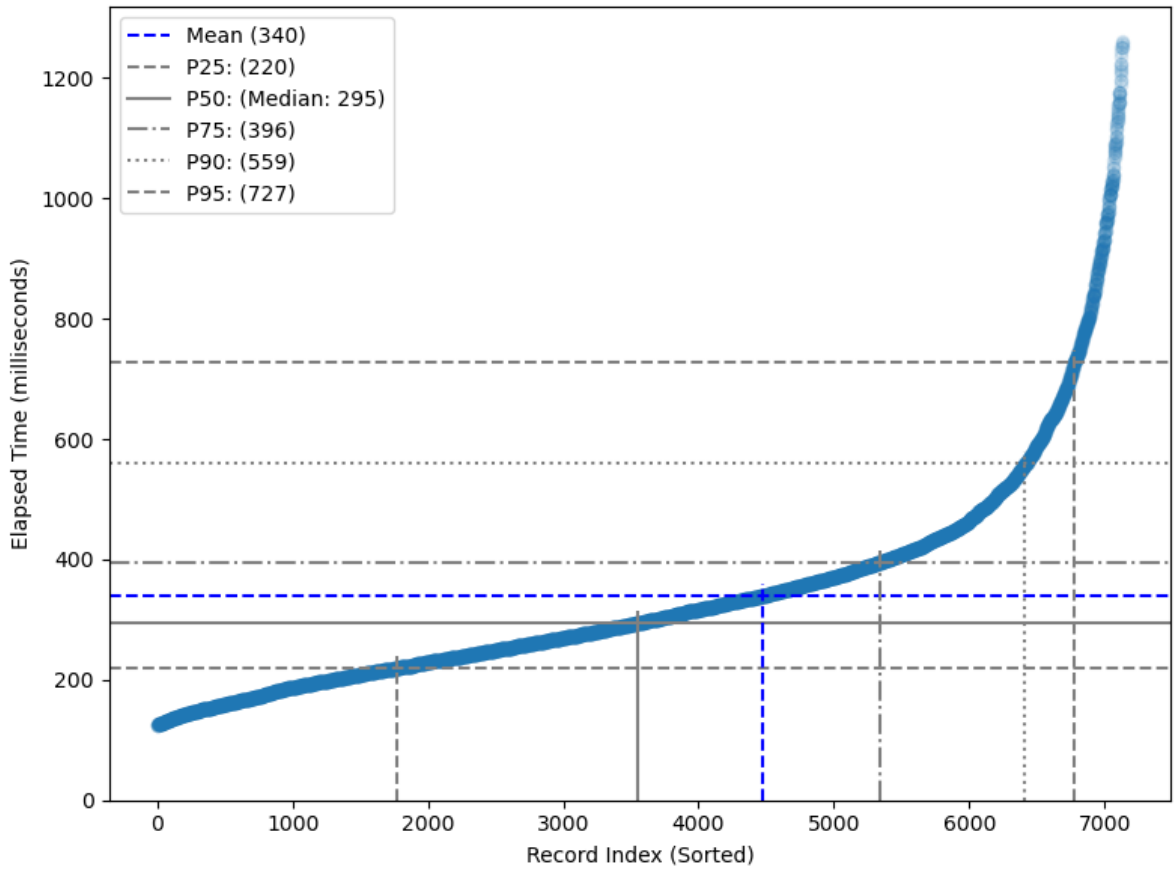

Figure 11: Downstream Service Sorted by elapsed time

Reviewing the data sequentially by time, by graphing it in a LOWESS diagram as in Fig. 12, the data shows a similar pattern to the same days as shown in the gateway service.

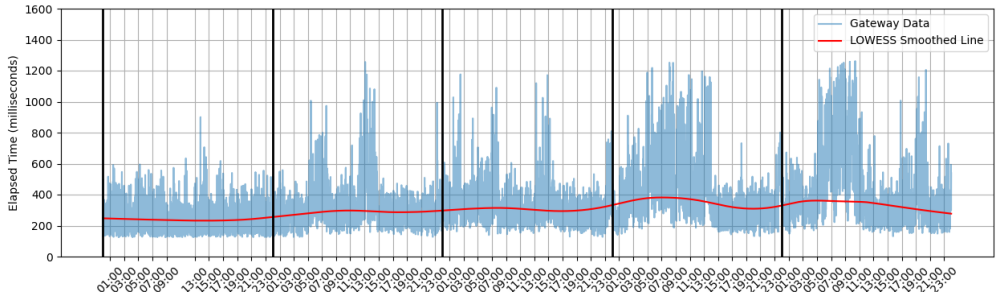

Figure 12: LOWESS Diagram Downstream

The same two days' worth of data were extracted to do a comparison that can be viewed in Fig. 13. These days were the second and third days of the five-day sequence and were chosen because they showed the highest fluctuation in performance. The goal of this comparison is to show how fluctuations in the downstream service affect the gateway service.

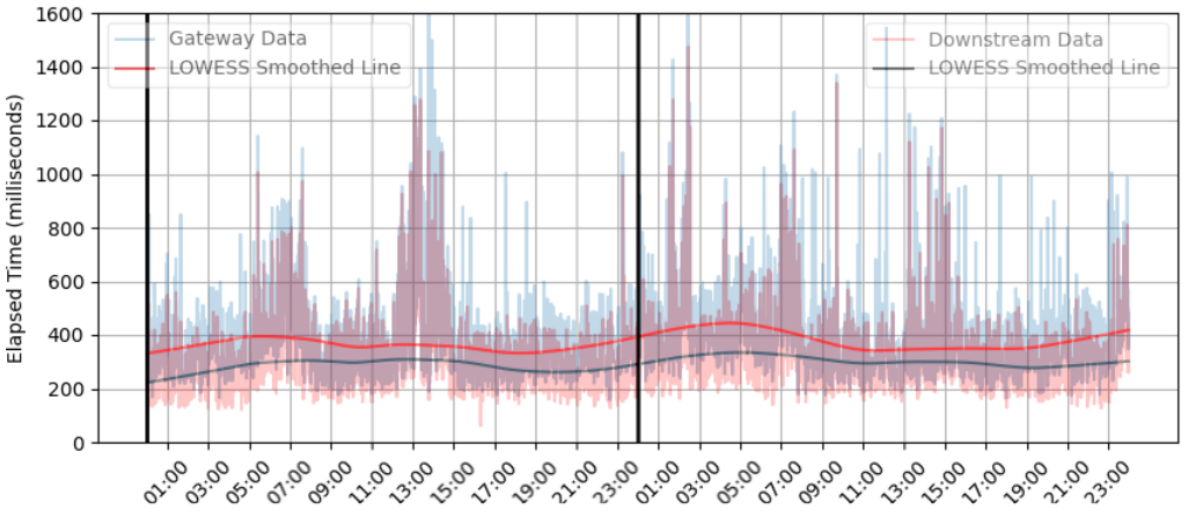

Figure 13: LOWESS Diagram Two Day Comparison

What can be observed in this two-day comparison is how similar the LOWESS line is between the two data sets, as well as the overall performance. There is an increase in latency in the gateway service, as there is a small amount of logic and networking calls involved that will delay these services. Based on tables 2 and 3, by reviewing the percentiles in these tables, there is a general delay of approximately 100 ms on average that the gateway service adds.

As noted earlier, the large performance spikes are regularly happening in non-peak hours, 01:00–07:00 hours. It was confirmed that there are processes that run overnight that impact the database, which degrades performance during these hours.

4.6.3 Modern Monolith Architecture

The same test was performed on the modern monolith architecture. The same requests were made to retrieve the same 20 employees of a company out of over 1,000 employees. This test was run over the same length of time and generated 20K+ data points to analyze. The overall breakdown of this table looks like table 4.

Metric	Completed Data Set	Trimmed Data Set
Mean	178 ms	179 ms
STD	72 ms	41 ms
Min	123 ms	125 ms
Max	7,704 ms	461 ms
P25	141 ms	141 ms
P50 (Median)	180 ms	180 ms
P75	188 ms	188 ms
P90	199 ms	198 ms
P95	228 ms	224 ms
P99	434 ms	401 ms
Data Points	20,059	19,857

Table 4: Microservice Metrics: Item List

The CV of this data:
Complete Data Set

$$CV = \frac{72}{178} \times 100\% = 40\% \tag{6}$$

Trimmed Data Set

$$CV = \frac{41}{179} \times 100\% = 23\% \tag{7}$$

We can see that the CV is much more realistic. We are seeing a CV value of 40% in the complete data set, and by trimming the top and bottom 0.5%, the CV comes down to 23%. The trimmed data set reflects a very stable service that we will see reflected in the graphs below.

What is driving those low CV numbers is the very low value for the STD. As the mean lowers in value, it is often harder to get a low STD. This represents a much faster and more reliable user experience compared to what the legacy microservices were able to achieve. Following a similar analysis as before, we can sort the data by elapsed time and view the sorted data in a graph; see Fig. 14. In this figure, a few attributes of this diagram are worth pointing out. The first thing to point out is that the significant rise of the tail happens after the 90th percentile; the curve is much sharper and is almost vertical. Whereas the microservice diagram in Fig. 5 begins a much steeper rise from the start of the line and begins a steep incline at the 75th percentile. The second unusual attribute of this diagram is the small but steep increase in performance that happens just before the 25th percentile line.

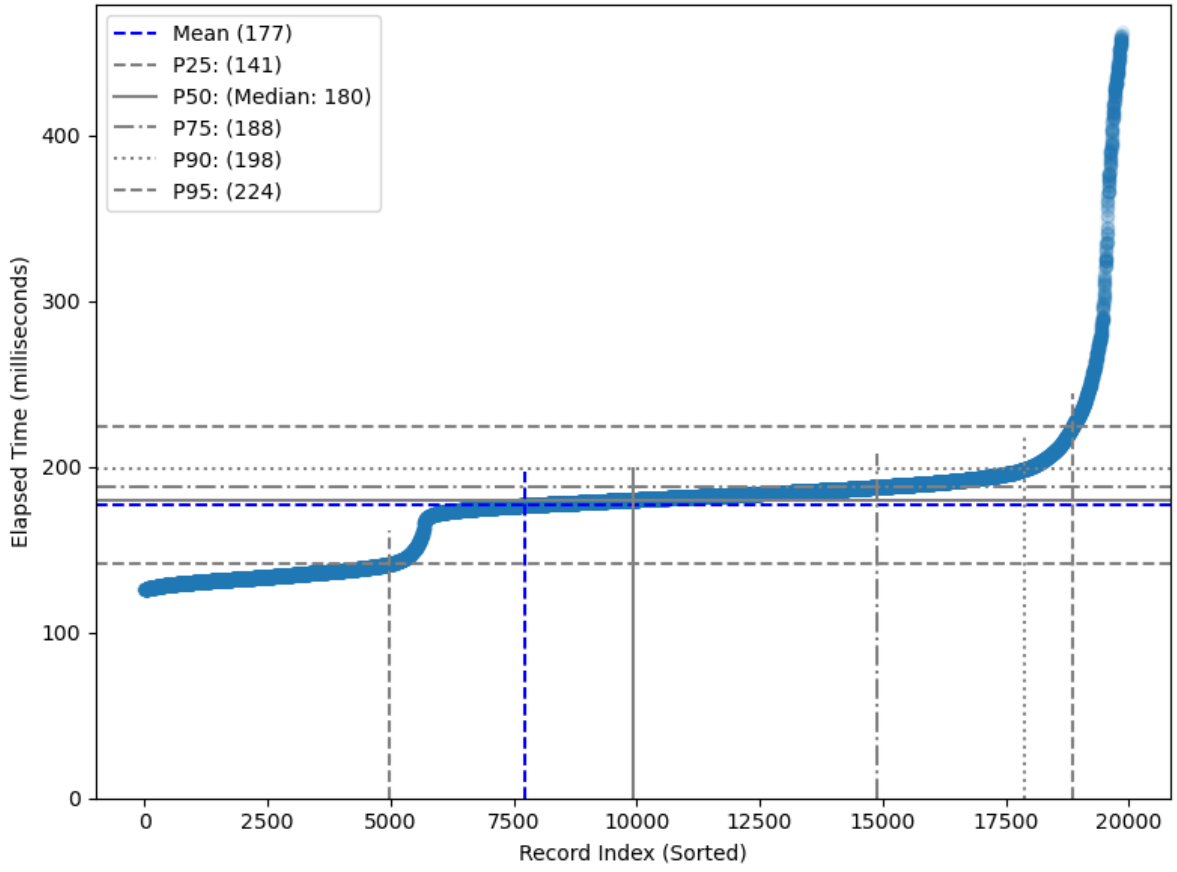

Figure 14: Modern Monolith Sorted by elapsed time

This spike is visible in the sorted view but is much more pronounced in the scatter diagram referenced in Fig. 15. What we see are distinct vertical bands or columns of data points, rather than a more continuous or diffuse spread as seen in Fig. 6. This is a surprising view of data for a couple of reasons. As stated before, a diagonal trend is expected in this type of diagram, showing a relationship between response size and time, and similarly to the microservices, that relationship is not there. The separate groups of data caused a deep dive into our monolithic architecture, examining the performance of the monolith, and eventually into the database. Working in a cloud-native environment, it was difficult to come to a clear understanding of what was causing the clear groupings of this data. Eventually, it was discovered that there is a level of caching at the database level that optimizes performance but is very volatile. When the cache was active, we saw extremely fast performance, whereas when the cache was not active, the performance was around 50 milliseconds slower—still quite fast, but noticeable in the measurements. Comparing the response time in Fig. 15 to the microservice scatter chart in Fig. 6, due to the higher performance and variability of the services, this 50-millisecond gap would not be visible.

On the left side of Fig. 15, the size of the response is displayed. It is worth mentioning that if

this diagram is compared with the similar diagram from the microservices (see Fig. 6), the size of the responses is very different. The microservice is returning approximately *twice* the amount of data as the monolith. In the microservice architecture, the services are interacting with a very generic API, and a generic dataset is retrieved that is much larger than what is needed to render the user interface. Meanwhile, the monolith architecture is tailored for that presentation, so the response size is a smaller dataset. To achieve a similar data load, a list of 40 items could have been returned in the monolith architecture, and after some experimentation, it was discovered that there was not a significant change in response time based on the size of the data being returned. With that information available, it was decided to compare similar functionality, not similar data transfer amounts.

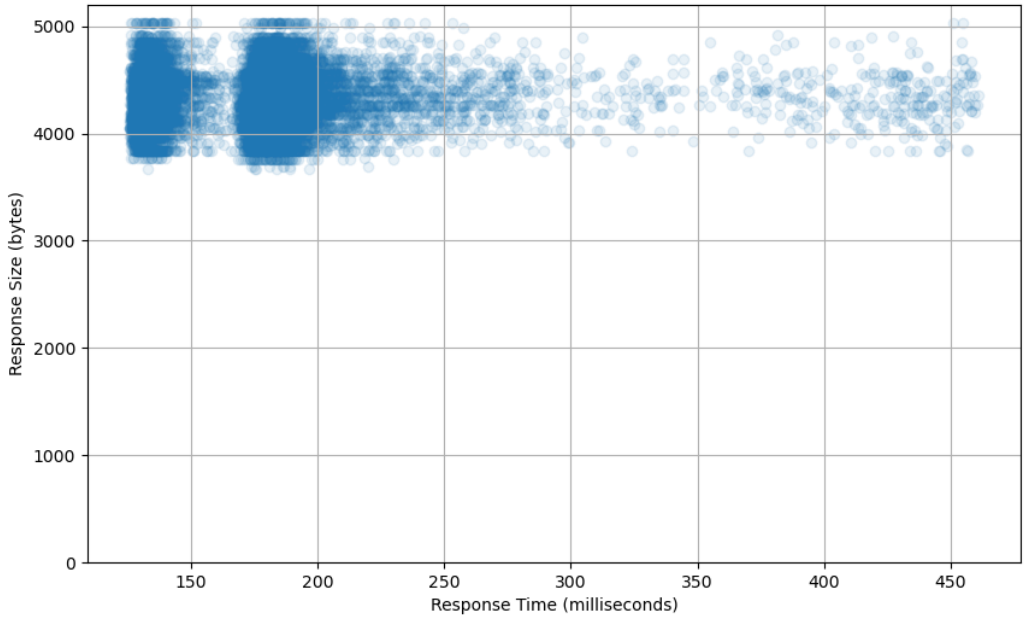

Figure 15: Monolith Scatter Plot Response Size and Response Time

To get a full picture of this database caching performance visualization, we can look at the wider dataset. The LOWESS diagram shown in figures 16, 17, 18, 19 is a good tool to visualize the data as it happened over time. What these diagram show are two distinct performance levels related to the database caching: one that is very fast, recording transactions between 120–160 milliseconds, and another where transactions are measured between 170 and 210 milliseconds. The LOWESS diagram also shows a third tier of transactions, although very infrequent, that run above 400 milliseconds. These infrequent but noticeable delays are related to the data architecture keeping the data in sync between the write and the read replicas.

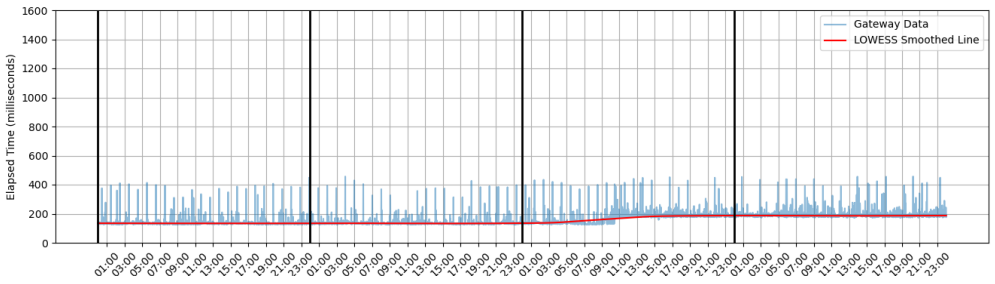

Figure 16: LOWESS Monolith Details Days 1-4

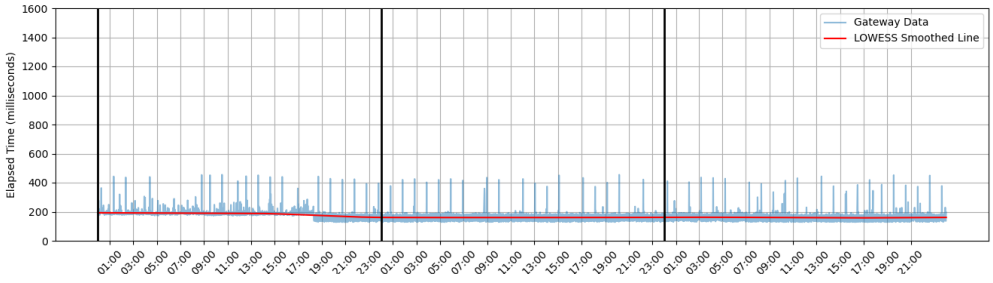

Figure 17: LOWESS Monolith Details Days 5-7

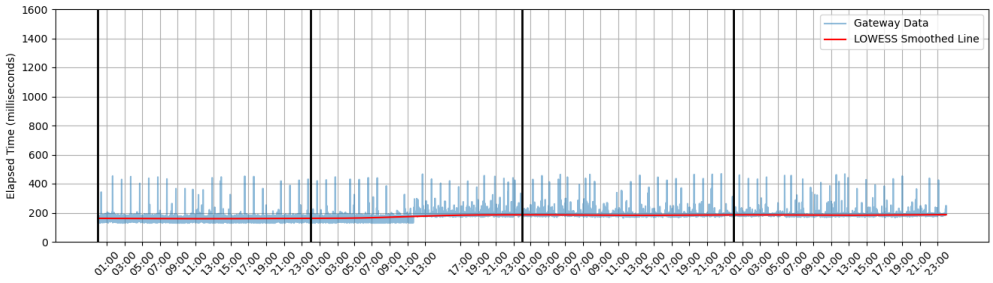

Figure 18: LOWESS Monolith Details Days 8-11

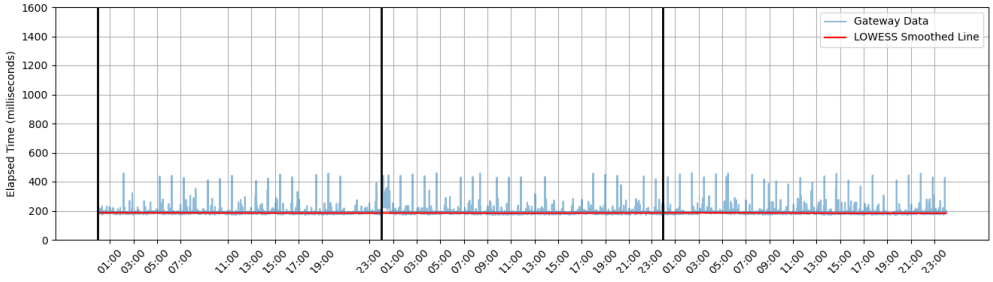

Figure 19: LOWESS Monolith — Days 12–14

Taking the findings of these three types of diagrams, two factors were identified that affected the CV value. The caching at the database tier causes some transactions to be extremely fast, and the syncing of data that happens between the read and write replicas creates a periodic delay that can cause transactions to take up to 300 milliseconds longer.

At the current levels of load that we are experiencing, the monolith architecture has an impressive level of performance compared to the microservices. Not only is it much faster, but it is also much more consistent and reliable in its performance. Performance typically ranges from 150 milliseconds to 500 milliseconds, compared to the microservices where performance ranges from 200 milliseconds to 1,222 milliseconds.

4.7 Item Details

The second API that was tested was to retrieve the details of a specific item. This endpoint hits the same gateway service but accesses a different endpoint. This service call is much more complicated than the simple list of items. Based on the configuration of the item being requested, multiple types of hierarchical data may be returned. In the microservice architecture, depending on the type of item being requested, a series of requests will be made to bring back all of the detailed information. Although the response size is smaller than the item list data, the multiple queries create more complexity and cause more latency in the overall system.

4.7.1 Microservice Legacy Architecture: Item Details

The microservice architecture was tested for 14 days, measuring the response time of a transaction every minute. Normally, there would be over 20K measurements for this data set; however, so many of the requests took longer than a minute to complete that it slowed down the accumulation of data, resulting in only 19,838 data points collected in the two-week period. Even though we had 322 fewer data points, there is still enough data to show the trends over the two-week period.

Metric	Complete Data Set	Trimmed Data Set
Mean	953 ms	943 ms
STD	445 ms	305 ms
Min	94 ms	434 ms
Max	23,102 ms	2,030 ms
P25	677 ms	679 ms
P50 (Median)	929 ms	930 ms
P75	1,152 ms	1,150 ms
P90	1,340 ms	1,332 ms
P95	1,500 ms	1,478 ms
P99	1,886 ms	1,779 ms
Data Points	19,838	19,638

Table 5: Microservice Metrics: Item Detail

The CV for this data is as follows:
Complete Data Set

$$CV = \frac{445}{953} \times 100\% = 47\% \tag{8}$$

Trimmed Data Set

$$CV = \frac{305}{943} \times 100\% = 32\% \tag{9}$$

In table 5, we see immediately that the performance on this call is much slower when compared to the list of items service. The mean value of the calls is 943 ms—almost a full second on average to make this request. With a mean measurement being so high, this allows more room for the STD to fluctuate, and in fact, there is a much better CV of 32% due to the very high mean value. When this situation occurs, we need to look deeper into the percentile values to understand the overall performance.

Looking at the 25th percentile, it signifies that 25% of the data points recorded were 679 milliseconds or less. Putting that in perspective, the 25th percentile of this service call is slower than the 90th percentile of the previous microservice call that retrieved a list of employees and is slower than the 99th percentile of the monolithic service to retrieve a list of employees. The remaining percentiles show a service that is performing poorly.

We can examine the sorted data plotted in Fig. 20. This diagram shows that the mean and median are almost identical, with a 13-millisecond difference. An important piece of data is the scale of the overall graph: the Y-axis ranges from about 500 milliseconds to over 2,000 milliseconds, with the 95% at 1,479 ms. With the mean and median both at over 900 ms, that signifies that half of the users who access this service experience a wait time of more than 0.5 seconds, and a small number wait as long as 2 seconds.

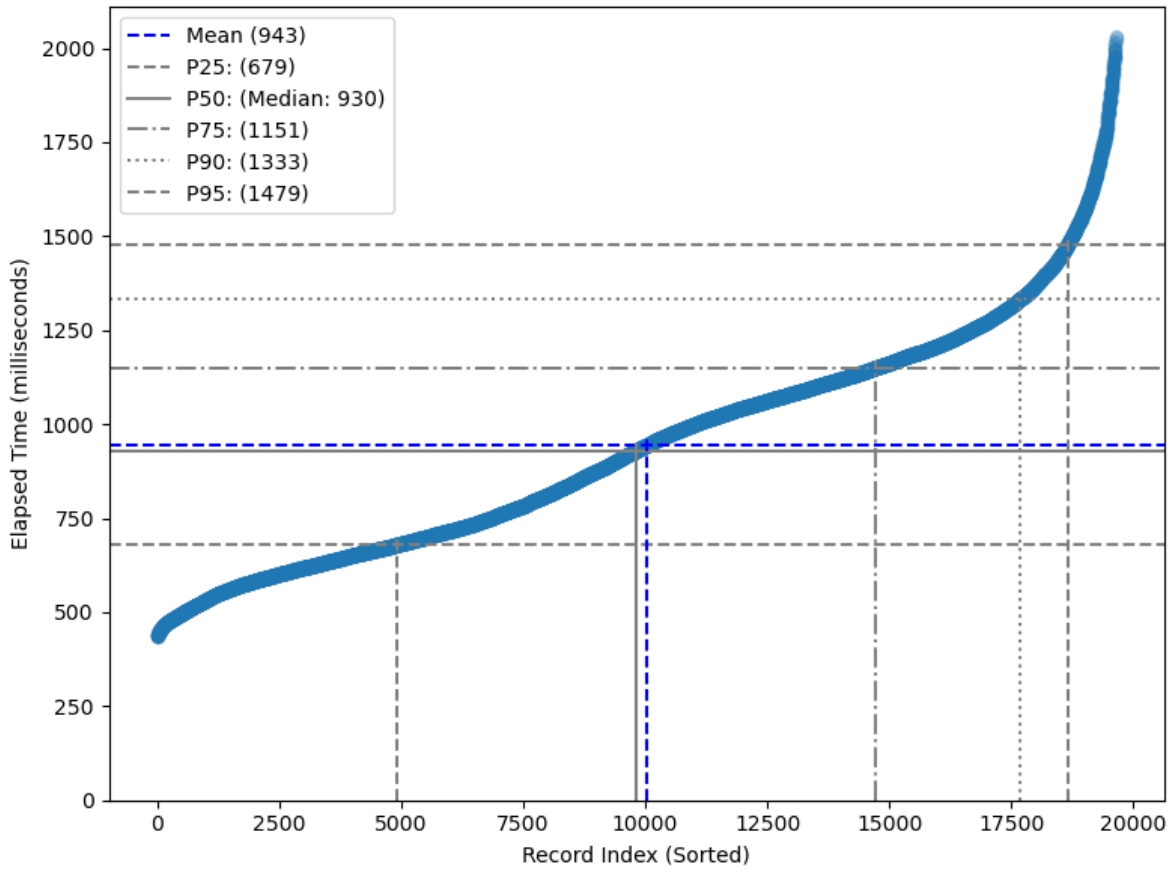

Figure 20: Sorted transactions for item details

The scatter diagram shows a different pattern of data. Where the corresponding Fig. 6 shows a well-distributed pattern of data, this version has multiple horizontal layers with very similar distribution patterns. What the diagram shows is that there is not a lot of variability in the response sizes; out of the 1,000+ employees tested, there were about eight different sizes of data that were returned. When graphed on the scatter diagram, the eight layers all took about the same amount of time.

For example, for the requests that were under 1,000 bytes, the range was from 400 ms to 1,300 ms. On the larger end of the spectrum, requests that were almost four times that size took the same amount of time. In a typical linear graph, as responses get larger, the response times would be slower, but that was not the case with this data set.

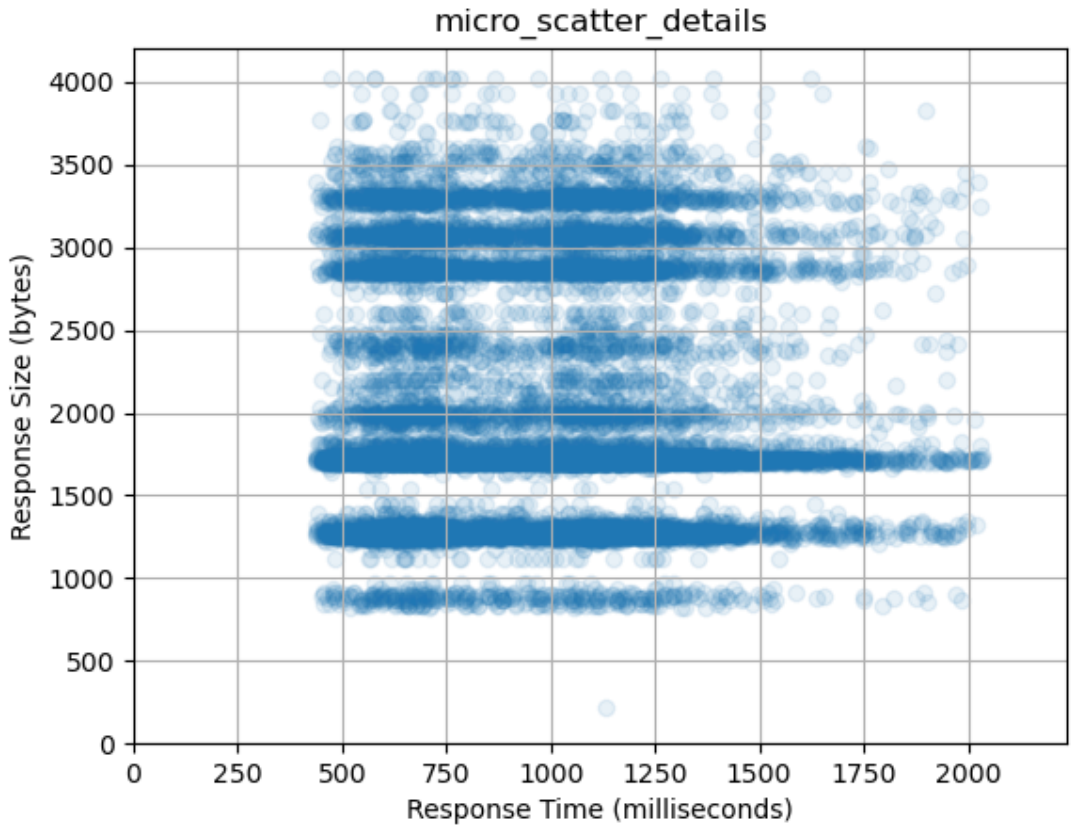

Figure 21: Micro Services Item Details

Looking at the LOWESS diagrams, we can see some spikes in performance, but for long periods of time, we see a consistent performance of around 1 second, and the thick bar signifies the high value of the standard deviation. Looking at these LOWESS diagrams, we can see that the performance spikes usually occur between 1:00 AM and 7:00 AM. This once again points to performance problems in the database that are affecting the overall performance of the entire system.

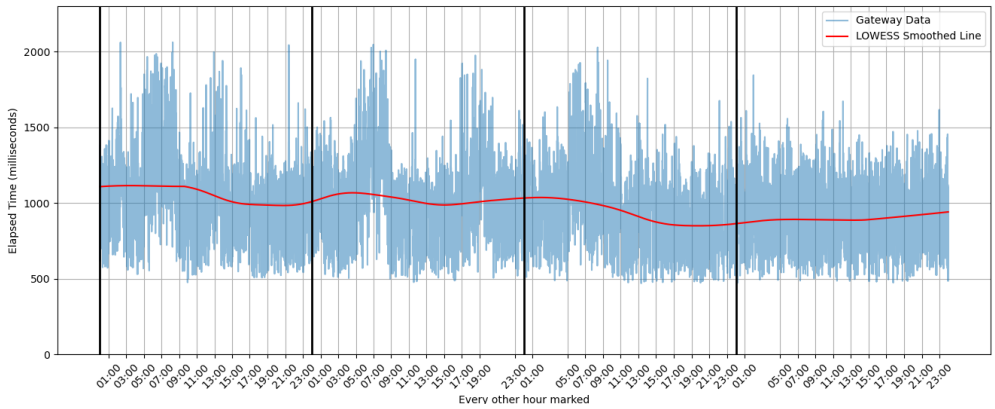

Figure 22: LOWESS Diagram Microservice: Item Details Days 1-4

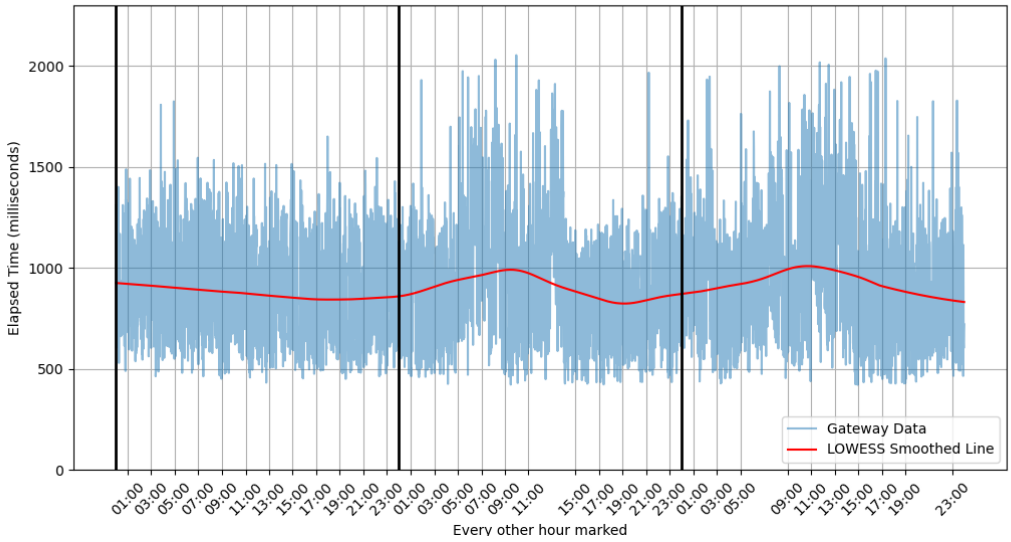

Figure 23: LOWESS Diagram Microservice: Item Details Days 5-7

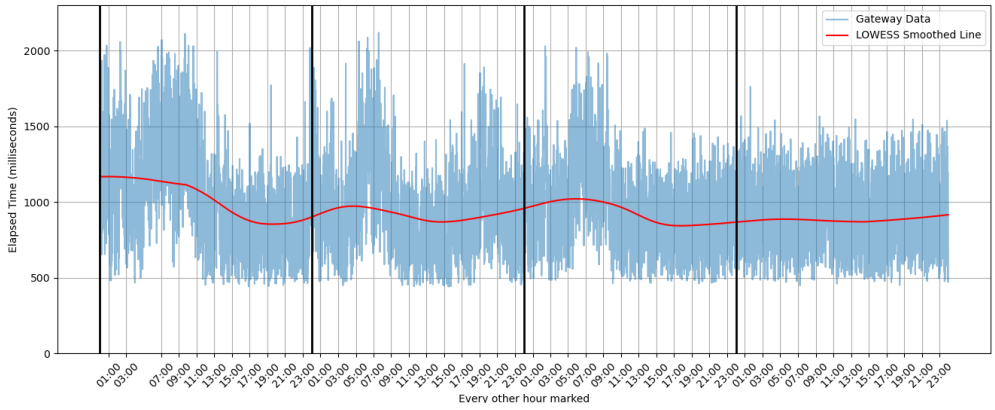

Figure 24: LOWESS Diagram Microservice: Item Details Days 8-11

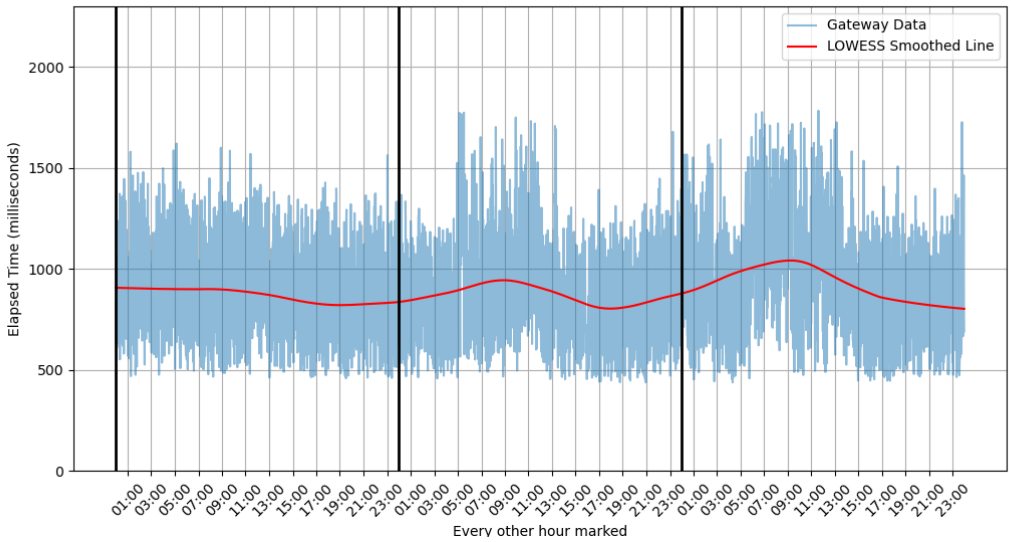

Figure 25: LOWESS Diagram Microservice: Item Details Days 12-14

The LOWESS diagrams are showing the wide variability in service calls overall. Additionally these diagrams help visualize the high performance spikes that are as high as 2 seconds while the best performance measurements are around half a second.

4.7.2 Modern Monolith Architecture: Item Details

The monolithic service architecture was tested against an API that returns the item details that a user requests. The metrics gathering was run for 14 days and generated 20K data points measuring response times. Similarly to the other tests, the service was measured every minute of every hour for this period of time.

The breakdown of this data is shown in table 6. Similarly to the other data sets, .5% was trimmed off of the slowest and fastest transactions. In the complete data set, the Mean value was 218 milliseconds with a standard deviation of 125 milliseconds, and in the trimmed data set, the Mean was 213 milliseconds while the STD was 52 milliseconds. Comparing this service call to the previous monolith service call, it is clear that retrieving the item details does take longer, as we saw in the legacy microservices. The difference in performance is around 30 milliseconds, but it was a measurable difference.

Metric	Complete Data Set	Trimmed Data
Mean	218 ms	213 ms
STD	125 ms	52 ms
P25	182 ms	182 ms
P50 (Median)	201 ms	201 ms
P75	226 ms	225 ms
P90	260 ms	258 ms
P95	305 ms	299 ms
P99	506 ms	450 ms
Data Points	23351	23117

Table 6: Monolith Architecture Metrics: Item Details

The CV for this data is as follows:
Complete Data Set

$$CV = \frac{125}{218} \times 100\% = 57\% \tag{10}$$

Trimmed Data Set

$$CV = \frac{52}{213} \times 100\% = 24\% \tag{11}$$

When describing the CV in equation 1, a strong-performing service should have a CV value of 50% or less, and preferably less than 20%. When looking at the complete data set, our CV is 57%, indicating there could be a performance challenge, while in the trimmed data set, the CV drops down to 24%. Looking at the percentiles for this data set, we see that the overall performance is quite strong, with 99% of the transactions around 500 milliseconds.

To better visualize these percentiles, we can look at the data points sorted by elapsed time; this is shown in Fig. 26. The scale on the left is important to look at first, as it sets the range of the data we are examining. Almost all transactions fall below 600 milliseconds, and the first three percentiles are close — within approximately 20 milliseconds of one another. This tight grouping is also indicative of the very small CV.

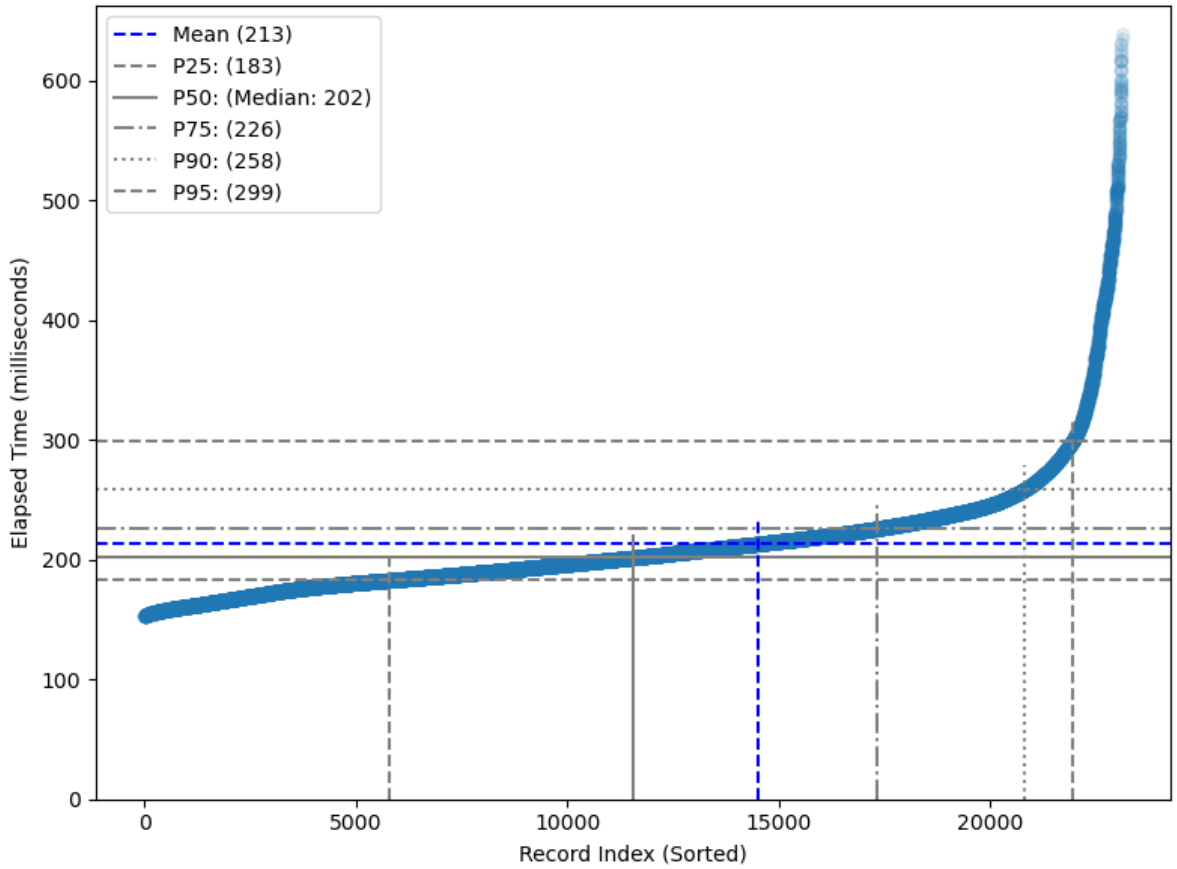

Figure 26: Monolith Item Details Sorted by Elapsed Time

Comparing the response size to the response time, we see once again that there is not a defined relationship between these two variables. The horizontal layering is showing up again, as in previous scatter diagrams, illustrating how responses of the same size take different times. We can see that the majority of calls are made between 100 milliseconds and 250 milliseconds, which is in alignment with our previous data visualizations.

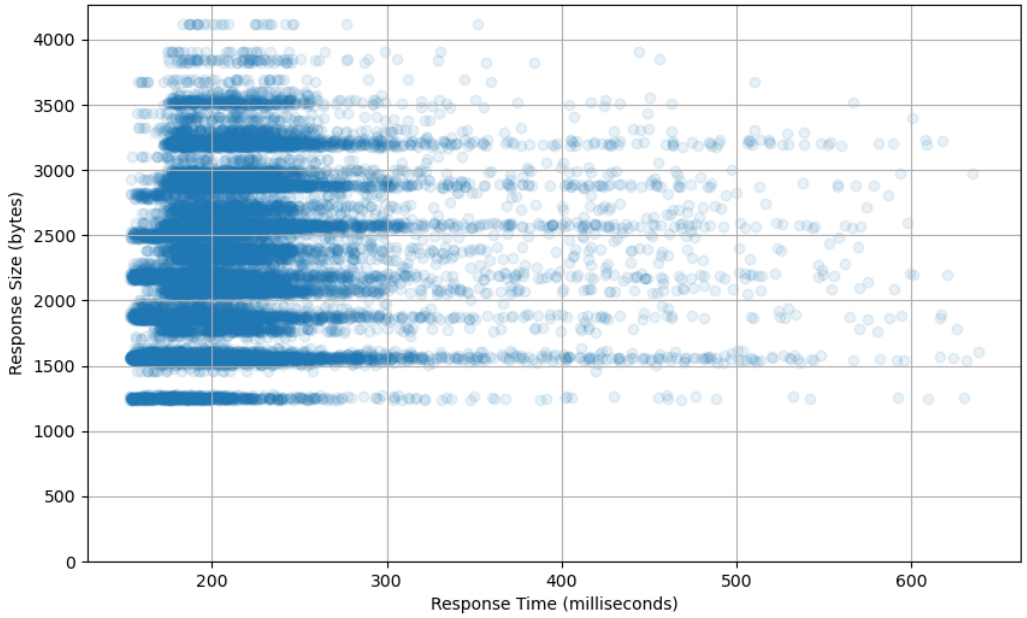

Figure 27: Monolith Item Details Scatter Diagram

If we graph this data into a LOWESS diagram to view the data over time, we can see a similar pattern to the monolith service that returned a list of items: there is a grouping of measurements that are extremely fast and another grouping where they are slightly slower. These two tiers are seen best in Fig. 28.

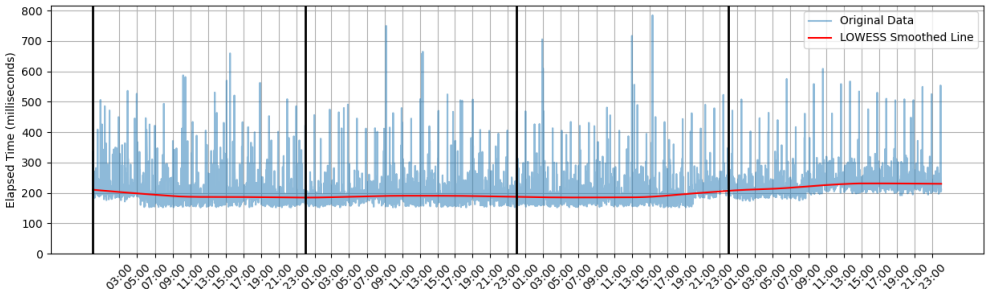

Figure 28: Monolith Item Details LOWESS Diagram Days 1-4

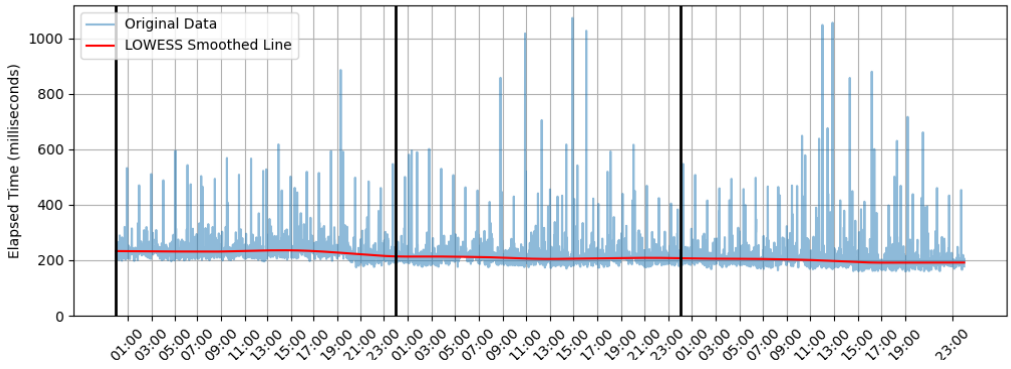

Figure 29: Monolith Item Details LOWESS Diagram Days 2-7

In the second week of the measurements, Fig. 30 and Fig. 31 show, based on the scale on the right, that the transactions had a more consistent range in that week. Based on a high number of transactions below the 200 millisecond range, it appears that the database was caching most of these measurements.

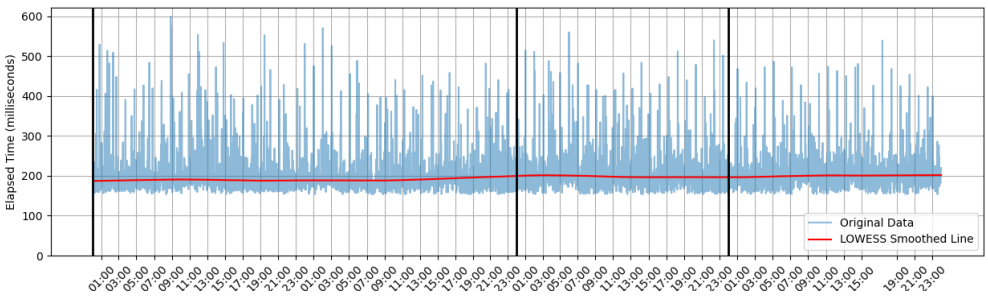

Figure 30: Monolith Item Details LOWESS Diagram Days 8-11

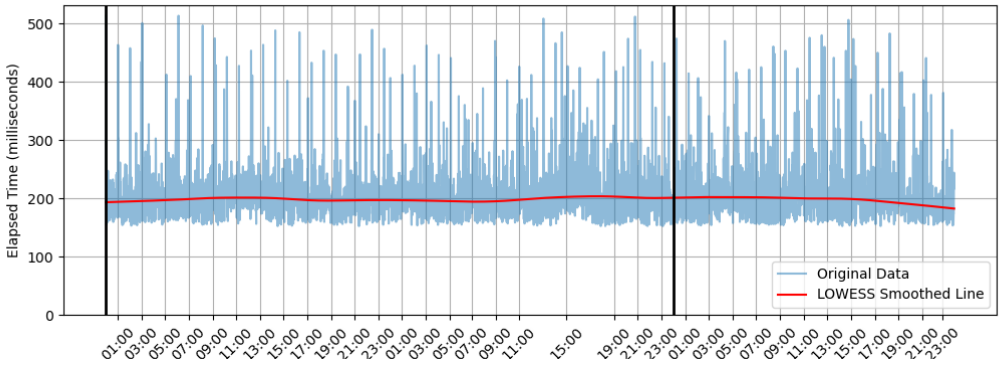

Figure 31: Monolith Item Details LOWESS Diagram Days 12-14

4.7.3 Scope Limitation

The original research plan included measuring and analyzing performance across more than two distinct services. However, due to the significant performance challenges observed in the legacy microservice architecture, expanding the scope of measurement would have primarily served to further highlight the disparity in data architectures rather than provide new insights into application architecture performance.

4.8 Discussion

In this use case study we analyzed two different services to answer research question 2:

> *What are the key metrics that would indicate that a modern monolith architecture should begin to move towards a microservice architecture?*

We started the study with two API services that were measured and analyzed for a microservice architecture that was originally built around 2012 and is currently running in production. Similar endpoints were measured and analyzed for a new monolithic architecture that is also running in production in parallel with the microservice architecture. What was discovered was that both systems were not experiencing any significant load on CPU, memory, or other resource constraints. It was clear that, based on the low resource usage of these systems, the legacy microservices are not performing at the level they should be. This study then became a deep dive into the specifics of the microservice architecture, looking to find the bottlenecks causing the performance degradation, and comparing the overall performance to the monolith service. What was discovered was that the database that the low-level microservices connect to has a lot of heavy usage at all times of the day, causing performance issues that are not easily solved by an application architecture. However, microservices can improve the overall resilience in this scenario, providing some level of stability.

The analysis of this data began with an assessment of the user load that is applied to these services. What was discovered is that there are approximately 6K users who access the application primarily between the hours of 8:00 AM and 6:00 PM EST, with spikes of users logging in throughout the day.

Reviewing the information that was shared in the Literature Review, if performance was the primary deciding factor on whether to adopt microservices, a daily load of 6,000 users would not likely be enough to merit the adoption of microservices. The Literature Review documented from multiple sources that a microservice architecture performs best with a high user base that generates a significant load on the services.

This microservice architecture was built to isolate and stabilize the user experience that was reliant on the database built for a centralized administration system. This source system is used by many business functions in this organization. Due to this high access and nightly batch jobs, the data architecture is not well suited to providing data to a web-based application. Adding multiple layers of services to encapsulate the database away from the services contributed to the inconsistency of the application by adding multiple network calls that have unpredictable response times.

In this specific use case, to improve the overall performance of these services, the integration with the source system database had to be decoupled. In the new architecture, a process was put in place to extract the data from the source system, transform the data into a format that could be more efficient for

these services, and provide this new formatted data to the monolith application. This was a successful effort, bringing the response times down from a mean of 436 milliseconds to 179 milliseconds, and the P99 transactions from 1,148 milliseconds to 401 milliseconds.

In the second set of APIs, even more extreme behaviors were observed. In the legacy microservice architecture, a series of calls are made to retrieve all of the necessary data to assemble the response. Each call to retrieve data runs through multiple services; once the data is retrieved, it is assembled and returned through the service. The mean for these calls in the legacy system is 943 milliseconds compared to 213 milliseconds for the monolithic service. The P99 transactions improved from 1,779 milliseconds to 450 milliseconds. This highlights the advantages of a custom data model versus building a solution around an API designed to serve multiple needs.

At the bottom of the graphs in figures 15 and 6, we can see the response times. We can compare the minimum and maximum values and observe that the monolith ranges from around 125 milliseconds to 461 milliseconds, while the microservices range from 188 milliseconds to almost 1,425 milliseconds. We can conclude that the monolith architecture is much faster and more consistent than the microservice architecture.

The consistent high performance and low STD of the monolith architecture can be explained by the simplified architecture. Every data request is made to a single consolidated database, providing fast and reliable performance. In comparison, requests to get data from the legacy system require a series of network calls to retrieve data from systems that were not designed for the specific functionality being implemented, thereby increasing the overhead on the network and impacting the performance of the database.

In summary, the microservice architecture is performing poorly and inconsistently due to database limitations. The overall performance of the application seems to be unaffected by the user load or any type of resource limitation. When comparing the gateway service and the downstream service, we can see that, on average, the gateway is adding about 100 milliseconds to the overall request. Using that number as a baseline and reviewing the architecture diagram in Fig. 1, there are four different microservices involved in a single request, each adding a variable amount of latency just for networking calls.

The monolithic architecture, for a similar user load, is performing much better. Having a simpler architecture and using modern technologies plays a part in the overall performance, especially around the consistency of the metrics gathered. The biggest performance gains are around building a data model that is decoupled from the source system.

Due to the relatively low daily users, as shown in Fig. 3, there was not enough traffic to find a threshold where we begin to see a performance degradation. With the services also having enough computing resources, a threshold in that area was also unable to be found. Overall, the intent was to find those thresholds in the use case, and that goal was not achieved. The study was able to confirm that the estimate provided by the research has some validity in its estimate; in paper [1], it was suggested that the threshold could be around 10–20K concurrent users based on the amount of resources a request takes.

5 Research Question 3: Common Design Patterns

The final research question to address in this study:

> *Are there common design patterns that can be leveraged by both architectures and how does the architecture change their implementation?*

A variety of articles were researched examining the best practices of how microservices are managed. The following categories were discovered in this research:

- Containerization and Orchestration
- Observability
- Continuous Integration and Deployment
- Scalability
- Resilience and Fault Tolerance
- Security

Although many of these categories have become industry standards, when working with legacy software it is surprising how many legacy systems do not support these design patterns. Based on the use case studied earlier, it is clear that there was some adoption of microservices prior to containerization, but containerization was an enabler for many companies to begin experimenting in that space. Once containerization and orchestration were adopted, each of the other design patterns quickly became standards, and each category could easily have its own study to deeply analyze best practices and benefits.

5.1 Containerization and Orchestration

The adoption of containerization and orchestration frameworks is a primary shared characteristic. This extends beyond just deploying, as it supports key features such as automatically scaling either the monolith or key aspects of a microservice so that the overall system can handle changes in user demand.

Containerization: Containerization with tools like Docker provides a lightweight and portable platform for companies to develop and deploy their applications on. Containers also provide a consistent, isolated environment that makes it easier for development teams to deliver a working solution, playing a large part in reducing the "it works on my machine" deployment problems [7].

Orchestration: Orchestration frameworks, such as Kubernetes, are responsible for managing the deployment, scaling, and operation of containers. Regardless of the architecture choice, orchestration frameworks perform the same function. These tools are also important in improving fault tolerance of managed containers by detecting their health and automatically restarting them as necessary [8].

5.2 Observability

One of the first challenges that microservices face is observability. Based on the paper *"Observability in Microservices"*, the author breaks down observability into three separate areas: Logging, Traceability, and Metrics [9].

- **Logging**: typically a simple text-based representation of events and errors that occur in a system.
- **Traces**: involve end-to-end visibility. This helps identify bottlenecks and failures.
- **Metrics**: gathering and recording metrics focused on overall health and system performance.

This paper specifically focuses on the tools and strategies to improve observability in containerized services. In an environment where a service is managed using a containerization tool and an orchestration framework, servers can be created and destroyed depending on load. This makes accessing log files directly unrealistic; therefore, this paper covers many categories of observability and points out the areas where challenges remain.

5.3 Continuous Integration and Deployment

The core principles of CI are frequent commits, automated builds, and automated testing. Continuous Deployment (CD) refers to the practice of automatically deploying every change that passes automated testing to production without requiring manual intervention [10]. Despite the differences in complexity and scale, both architectural solutions benefit from the same CI/CD principles and best practices.

- **Automated Testing**: Test automation is perhaps one of the most significant areas of automation in the development process. Manual testing is time-consuming and prone to human error, whereas automated testing accelerates the process and ensures more reliable results. Tools such as JUnit, Selenium, and Cypress allow teams to write tests that automatically validate code functionality as part of the CI/CD pipeline [10]. To build and deploy applications in a continuous CI/CD environment, automated regression testing and validation are critical regardless of the architecture. Microservices require testing across individual services and verification of system-wide reliability, while monoliths require robust unit and integration tests.
- **Progressive Delivery**: Also called feature flags or A/B deployments, this technique enables CI/CD pipelines by allowing features to be deployed but not made available to users until enabled. This reduces the risk of production outages and allows for an immediate rollback if the new version proves unstable [11].

5.4 Scalability

Scalability is the ability to adapt to fluctuating user demands and traffic loads. This is a fundamental need for any modern application. Traditionally, monolith architectures were limited to vertical scaling — increasing the hardware capacity of a single server. A modern monolith deployed using containers and an orchestration framework can scale horizontally. This is similar to a microservice, with the main difference being that the entire application is scaled, while in a microservice architecture one small service may scale independently to accommodate traffic increases.

Regardless of the architecture pattern, there are key design patterns that need to be implemented to properly scale:

- **Stateless Application Design**: Each request must contain the necessary information to perform the requested function, including security. In a containerized environment, applications cannot rely on a server to maintain state. As containers are destroyed and re-instantiated, any state maintained on a server would be lost.
- **Distributed Caching**: To reduce unnecessary, expensive database queries, caching is often implemented to increase performance. Caches must be expected to be volatile, as they may be destroyed at any time based on system needs.

5.5 Resilience and Fault Tolerance

Fault tolerance is defined as the ability to continue operating while some components fail. Fault tolerance improves overall system resilience. Resilient systems can absorb failures, recover from unexpected errors, and prevent outages. A poorly designed microservice can be as fragile as a poorly designed monolith. There are key principles and patterns that can be implemented to increase fault tolerance and resilience in both architectures. Containerization and orchestration frameworks largely improve resilience by supporting redundancy, failover, and replication. In addition, patterns such as circuit breakers, timeouts, retry mechanisms, bulkhead isolation, and graceful degradation, when implemented correctly, help create systems that are both resilient and fault tolerant.

5.6 Security

Security is a significant concern for any application deployed today. Regardless of the underlying architecture, security is a top-tier concern. As noted earlier, the wider surface area of a microservice architecture requires more effort to secure [1].

Applications built in either architecture pattern should follow security best practices at every step of the application lifecycle — from initial design and coding through testing, deployment, and operations.

6 Threats To Validity

Threats to validity represent factors that can undermine the accuracy and reliability of the research findings. Common threats in studies often stem from selection bias, data extraction bias, and conclusion bias. The following describes how these concerns were addressed in this study.

6.1 Selection Bias

In this study's context, selection bias could occur by choosing certain windows of time to collect data in a way that skews results. By selecting a consistent 14-day window for data collection, we ensured that each service had comparable, time-bound scenarios to reflect, minimizing this bias.

6.2 Extraction Bias

Extraction bias can occur when the data collected is not representative, such as repeatedly running common queries that create cached results and artificially improve performance. To address this risk, a sufficiently large data set was chosen so that each service retrieved only a small portion of it, reducing the influence of server-level caching or application optimizations. However, it was identified and acknowledged that some level of caching occurred at the database level, which could not be controlled. The observed overall impact was an improvement of approximately 50 milliseconds for certain requests in the monolith architecture.

6.3 Conclusion Bias

Conclusion bias is often related to starting a study with strong preconceived opinions and seeking data to confirm them. To mitigate this, a wide literature review was conducted to understand the pros and cons of each architecture. In addition, discussions were held to analyze the results from multiple perspectives. These discussions helped verify the conclusions drawn from the data.

7 Conclusion

In this study, a thorough literature review was conducted to contrast the benefits of microservice and monolithic architectures. The review indicates that there is a place for both patterns: microservices provide a level of scalability and resilience that some applications require, while many applications—particularly those with smaller user bases—perform more effectively with the simplicity of a monolith. For such applications, a "modern monolith" that incorporates contemporary deployment and scalability practices can offer excellent performance with reduced operational overhead.

The results from the use case highlight that performance improvements were driven primarily by data architecture choices rather than the application architecture alone. The latency introduced at multiple layers of a microservice-based solution was clearly visible, while the monolith demonstrated consistently lower response times due to its simplified request path and decoupled data model. These findings reinforce the importance of evaluating both data and application architecture when considering system performance and scalability.

In summary, the study sought to answer three research questions:

1. *By modifying the definition of a monolith, are there lessons from microservice architecture that could improve scalability and resilience?* The research supports the idea that adopting selected microservice design patterns—such as containerization, orchestration, observability, and automated deployment—can significantly improve the scalability and resilience of a monolithic system. When applied to a "modern monolith," these practices extend the benefits of microservices while maintaining the simplicity and lower operational overhead of a single deployable unit.

2. *What are the key metrics indicating when a modern monolith should transition to a microservice architecture?* While the literature review identified metrics such as concurrent user load, service scaling patterns, and network latency as metrics for deciding when a modern monolith may need to transition to a microservice architecture, the use case did not produce enough variation in these areas to define a precise threshold. The results instead showed that the performance and scalability of the systems were driven more by the design of the data model than by the application architecture itself. This shows that if a transition should occur it would be highly dependent on the context of the application and is best analyzed on a system that is showing signs of application server performance challenges not database performance concerns.

3. *Are there common design patterns that can be leveraged by both architectures, and how does the architecture affect their implementation?* The literature review confirmed that many industry-standard design patterns—such as containerization and orchestration, observability, continuous integration and deployment, scalability strategies, resilience and fault tolerance mechanisms, and security best practices—are applicable to both monolithic and microservice architectures. The core principles remain the same, but the scope and complexity of implementation differ: in a microservice architecture, these patterns must be applied across multiple deployed services, while in a monolith they can often be implemented on one server/container, reducing operational complexity.

Ultimately, the study confirmed that while microservices can offer advantages at scale, many applications benefit more from the simplicity, predictability, and lower overhead of a modern monolith—particularly when paired with a well-designed, decoupled data architecture.

7.1 Additional Observations

While the initial research plan included evaluating a broader set of services, the performance limitations of the legacy microservice architecture shifted the focus of the analysis. Continuing to analyze these services would have reinforced the differences between the underlying data architectures instead of providing new insights into the application architectures. The decision was made to measure and analyze the two selected services, allowing for a more thorough examination of the collected data.

A Appendix A: LOWESS Diagram

LOWESS stands for **LO**cally **WE**ighted **S**catterplot **S**moothing. Rather than fitting a single curve to all the data at once, LOWESS focuses on small "neighborhoods" of data points. For each point where a smooth line is desired, LOWESS examines the surrounding points, assigning more weight to those closer and less weight to those farther away. Within each neighborhood, it performs a simple regression—often a straight line or gentle curve—using these weighted points to determine the position of the smoothed line at that location.

The process is repeated for many points across the dataset, and the resulting predictions are connected to form a smooth, continuous curve. This approach highlights the underlying trends in the data while reducing the influence of noise and without assuming a fixed functional form.

References

[1] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022. doi: 10.1109/ACCESS.2022.3152803.

[2] Martin Fowler. Monolith first. `https://martinfowler.com/bliki/MonolithFirst.html`, 2015. Published: 2015-06-03.

[3] Nicholas Bjørndal, Antonio Bucchiarone, Manuel Mazzara, Nicola Dragoni, Schahram Dustdar, Fondazione Bruno Kessler, and T Wien. Migration from monolith to microservices: Benchmarking a case study. *Tech. Rep.*, 2020.

[4] Miloš Milić and Dragana Makajić-Nikolić. Development of a quality-based model for software architecture optimization: a case study of monolith and microservice architectures. *Symmetry*, 14(9): 1824, 2022.

[5] Nahid Nawal Nayim, Ayan Karmakar, Md Razu Ahmed, Mohammed Saifuddin, and Md Humayun Kabir. Performance evaluation of monolithic and microservice architecture for an e-commerce startup. In *2023 26th International Conference on Computer and Information Technology (ICCIT)*, pages 1–5. IEEE, 2023.

[6] Sanghamitra Nemmini, S Abhishek, T Anjali, and Sincy Raj. Fortifying information security: Security implications of microservice and monolithic architectures. In *2023 16th International Conference on Security of Information and Networks (SIN)*, pages 1–6. IEEE, 2023.

[7] Venkata Ramana Gudelli. Containerization technologies: Ecr and docker for microservices architecture. *International Journal of Innovative Research in Management, Pharmacy and Sciences (IJIRMPS)*, 11(3), 2023.

[8] Er Vishesh Narendra Pamadi, Shakeb Khan, Er Om Goel, et al. A comparative study on enhancing container management with kubernetes. *International Journal of Advanced Research and Interdisciplinary Scientific Endeavours*, 1(3):116–133, 2024.

[9] Ummay Faseeha, Hassan Jamil Syed, Fahad Samad, Sehar Zehra, and Hamza Ahmed. Observability in microservices: An in-depth exploration of frameworks, challenges, and deployment paradigms. *IEEE Access*, 13:72011–72039, 2025. doi: 10.1109/ACCESS.2025.3562125.

[10] Vincent Uchenna Ugwueze and Joseph Nnaemeka Chukwunweike. Continuous integration and deployment strategies for streamlined devops in software engineering and application delivery. *Int J Comput Appl Technol Res*, 14(1):1–24, 2024.

[11] Cosmin-Ioan Roşu and Mihai Togan. A modern paradigm for effective software development: Feature toggle systems. In *2023 15th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–6, 2023. doi: 10.1109/ECAI58194.2023.10193936.