UMEÅ UNIVERSITY

Evaluating Cost Efficiency in Scaling Software Architectures:

A Comparative Study of Vertical and Horizontal Scaling Approaches in Financial Workflows

Oskar Westerlund Holmgren
May 2025

Examiner: *Henrik Björklund*
Academic Advisor: *Nayereh Rasouli*
External Advisor: *Simon Bonér*

Master of Science Programme in Computing Science and Engineering, 300 Credits

Abstract

This thesis investigates the cost efficiency of vertical and horizontal scaling approaches in financial software architectures by implementing and evaluating equivalent monolithic and microservice versions of a high-throughput brokerage system. The research examines what bottlenecks limit effective scaling in each architecture and identifies when a monolithic architecture with vertical scaling should be chosen over a microservice architecture with horizontal scaling based on performance-to-cost considerations.

The study implements a financial system handling both IO-intensive order processing and CPU-intensive price update workflows commonly found in brokerage systems. Both architectures are deployed in a Kubernetes environment with equivalent resources. Both incorporating realistic security features like mutual TLS encryption, and are subjected to identical workloads.

Results reveal that neither architecture universally outperforms the other across all scenarios. The monolithic architecture demonstrates significantly higher initial order throughput but struggles to effectively scale both workflows simultaneously, with performance gains heavily dependent on database resources rather than system resources. The microservice architecture, while starting with much lower throughput, scales both workflows more effectively and continuously, but never catches up with the monolithic order processing capability. Service mesh technology significantly impacts microservice performance by adding substantial communication latency and resource consumption.

The findings suggest that monoliths are better suited for simple systems with single dataflows, particularly those leveraging complex database operations, while microservices offer better resource control for multi-dataflow systems despite communication overhead. This research contributes valuable insights for architects and developers designing scalable cloud-native systems, highlighting that architectural decisions should consider specific workflow characteristics rather than following universal best practices.

Acknowledgment

This thesis represents not only my individual work but also the culmination of guidance, support, and encouragement from many individuals who have contributed to this journey in various ways. I would like to take this opportunity to express my gratitude to everyone who has supported me throughout this research process.

I would like to express my sincere gratitude to my academic advisor at Umeå University, Nayereh Rasouli, for her invaluable guidance, expertise, and continuous support throughout this research journey. Her insights and feedback have been instrumental in shaping this thesis.

I am deeply grateful to Simon Bonér, my advisor at Clear Street, for his mentorship and technical expertise that greatly contributed to the practical aspects of this research. I also would like to extend my appreciation to the entire Clear Street team for providing me with the opportunity to conduct this research within their organization, as well as for their technical support and industry insights that enhanced the real-world applicability of this study.

My heartfelt thanks go to my family and friends for their unwavering encouragement and patience during this challenging period. Their constant support, moral encouragement, and much-needed distractions when the research process became overwhelming have been a source of strength throughout my academic journey.

This thesis would not have been possible without the collective support of all these individuals.

Contents

1 Introduction **1**
1.1 Research Questions . . . 1
1.2 Thesis Outline . . . 2

2 Background **3**
2.1 Distributed Systems . . . 3
2.2 Cloud Computing . . . 3
2.2.1 Cloud Native . . . 4
2.2.2 Amazon Web Service . . . 4
2.3 Software Architectures . . . 4
2.3.1 Monolith . . . 4
2.3.2 Microservice . . . 5
2.4 Scaling . . . 5
2.4.1 Vertical Scaling . . . 5
2.4.2 Horizontal Scaling . . . 6
2.4.3 Load Balancers . . . 6
2.5 Kafka . . . 6
2.5.1 Partitions . . . 6
2.6 Kubernetes . . . 6
2.6.1 Istio . . . 7
2.7 Financial Background . . . 7
2.7.1 Financial Terms . . . 7
2.7.2 Financial Systems . . . 8

3 Related Work **9**

4 Method **10**
4.1 System Overview . . . 10
4.2 Datasets . . . 11
4.2.1 Instruments . . . 11
4.2.2 Orders . . . 12
4.2.3 Start Prices . . . 13
4.2.4 Price Updates . . . 13
4.3 Storage . . . 14
4.3.1 SQL Optimization . . . 15
4.4 Monolith Design . . . 15
4.4.1 Order Processing . . . 15
4.4.2 Price Processing . . . 17
4.5 Microservice Design . . . 17
4.5.1 Order Processing . . . 17
4.5.2 Price Processing . . . 19
4.6 Infrastructure . . . 20
4.6.1 Storage Setup . . . 20
4.6.2 Kafka Setup . . . 21
4.6.3 Istio Setup . . . 21

4.7 System Configurations . . . 21
4.7.1 Monolith . . . 22
4.7.2 Microservice . . . 22
4.8 Measurement Definition . . . 23
4.8.1 System Load . . . 23
4.8.2 Profilers . . . 24
4.9 Resources When Scaling . . . 24
4.9.1 Underlying System Nodes . . . 25
4.9.2 Database Nodes . . . 25
4.9.3 Monolith Resource Usage . . . 25
4.9.4 Microservice Resource Usage . . . 26

5 Result **27**
5.1 Discovering Bottlenecks(RQ0) . . . 27
5.1.1 Performance Tuning of the Monolith . . . 27
5.1.2 Performance Tuning of the Microservice . . . 29
5.1.3 Scaling Monolith . . . 35
5.1.4 Scaling Microservice . . . 40
5.2 Comparing Scalability(RQ1) . . . 45

6 Discussion **48**
6.1 Discovering Bottlenecks(RQ0) . . . 48
6.1.1 Monolithic Architecture . . . 48
6.1.2 Microservice Architecture . . . 49
6.2 Comparing Scalability(RQ1) . . . 49

7 Conclusion & Future Work **52**
7.1 Future Work . . . 52
7.1.1 Caches . . . 53
7.1.2 Monolith & Sharding . . . 53
7.1.3 Increase Scale . . . 53
7.1.4 Partitioning Database . . . 53
7.1.5 Hybrid Architectures . . . 53

Appendix A PostgreSQL Configuration files **57**
A.1 PostgreSQL Node 1 . . . 57
A.2 PostgreSQL Node 3 . . . 57
A.3 PostgreSQL Node 3 . . . 58

List of Figures

1 System overview, order and price update dataflows 11
2 Wiener Process, used to generate price updates in instruments. 14
3 Order validation at Monolith (Sequence Diagram) 16
4 Order processing at Monolith (Sequence Diagram) 16
5 Price update processing at Monolith (Sequence Diagram) 17
6 Order validation at Microservice (Sequence Diagram) 18
7 Order processing at Microservice (Sequence Diagram) 19
8 Price update processing at Microservice (Sequence Diagram) 19
9 Investigating the effect of the number of price threads on Order & Unrealized P&L Throughput. Running on system node **S1** and database node **D1**. . . . 28
10 Investigating the effect of the number of order threads per vCPU on Order & Unrealized P&L Throughput. Running on system node **S1** and database node **D1**. 29
11 Investigating the difference between configurations from Tables 10 & 9 on Order & Unrealized P&L Throughput. Running on system node **S1** and database node **D1**. 32
12 Investigating the difference between configurations from Tables 11 & 12 on Order & Unrealized P&L Throughput. Running on system node **S1** and database node **D1**. 33
13 Investigating the effect of using threads in the order consumers on Order & Unrealized P&L Throughput. Running on system node **S1** and database node **D1**. 34
14 Investigating the effect of scaling the Monolith with database node **D1** on the Order & Unrealized P&L Throughputs. 36
15 Investigating the effect of scaling the Monolith with database node **D2** on the Order & Unrealized P&L Throughputs. 37
16 Investigating the effect of scaling the Monolith with database node **D3** on the Order & Unrealized P&L Throughputs. 38
17 Order Throughput when scaling the monolith. Includes measurements for each database node. 39
18 Unrealized P&L Throughput when scaling the monolith. Includes measurements for each database node. 39
19 Investigating the effect of scaling the Microservice with database node **D1** on the Order & Unrealized P&L Throughputs. 41
20 Investigating the effect of scaling the Microservice with database node **D2** on the Order & Unrealized P&L Throughputs. 42
21 Investigating the effect of scaling the Microservice with database node **D3** on the Order & Unrealized P&L Throughputs. 42
22 Order Throughput when scaling the microservice. Includes measurements for each database node. 43
23 Unrealized P&L Throughput when scaling the microservice. Includes measurements for each database node. 44
24 The Order Throughput of the monolith and microservice, mapped to the hourly cost of running the systems. 45

25 The percentage increase of the Order Throughput of the monolith and microservice, from initial scaling step (Node 1), mapped to the hourly cost of running the systems. 46
26 The Unrealized P&L Throughput of the monolith and microservice mapped to the hourly cost of running the systems. 47
27 The percentage increase of the Unrealized P&L Throughput of the monolith and microservice, from initial scaling step (Node 1), mapped to the hourly cost of running the systems. 47

List of Tables

1 Kafka Resource Specification 21
2 Pod instance count used in the microservice at the first scaling step 22
3 Resource for pods used in the microservice at the first scaling step 23
4 Underlying System Nodes, which the monolith and microservice will run on . 25
5 Available database nodes, that is, nodes on which the PostgreSQL database can be run 25
6 Instance count for the microservice configurations used during the scaling . . 26
7 Resources for pods used by the microservice configurations during the scaling 26
8 Average CPU usage of Monolith & Database, when running a Monolith with 7 order threads and 1, 3 and 7 price threads. Running on system node **S1** and database node **D1**. 28
9 Instance count for each service, in four different configurations, when investigating increasing resources given to order and price consumers. Using database node **D1**. 31
10 Resources for each service, in four different configurations, when investigating increasing resources given to order and price consumers. Using database node **D1**. 31
11 Instance count for each service, in two different configurations, when investigating the usage of fewer pods. Using database node **D1**. 32
12 Resources for each service, in two different configurations, when investigating the usage of fewer pods. Using database node **D1**. 33
13 Database TCP Latency and CPU utilization for databases **D1** and **D2**, when scaling the monolith. 40

1 Introduction

Cloud computing has become a vital part of today's industry, with industry revenue increasing each year [7]. It can provide great economic benefits for companies, such as lower operational and infrastructure costs. For example, companies do not have to acquire their own IT infrastructure or experienced personnel to run it, resulting in a more agile workflow[2]. Furthermore, since it also allows companies to pay only for the resources they use, they can adapt and scale their operations based on demand[2].

To efficiently use cloud computing and benefit from not running their own infrastructure, companies need to use the resources they pay for efficiently. This means, amongst other things, efficient handling of a growing workload by adding additional resources to your systems and removing excess resources or "scaling" your resources based on the demand[5]. To accomplish this, you can either use "vertical" or "horizontal" scaling. The choice of scaling approach also comes with limitations and demands on the software architecture to support it [38].

Over the years, more organizations have started using the microservice architecture when developing their software [25], [14]. Mentioning that the breaking of a monolith into a microservice had clear engineering benefits, including improved flexibility, simplified scaling, and easier management [25]. Since the microservice architecture makes a system "ideal" for the horizontal scaling approach [9], this also has resulted in a change in the scaling approach chosen.

Then a discussion could be held about whether this shift is always the best way forward or whether these systems could benefit from a "monolithic" architecture and vertical scaling. For example, the "Microservice Premium" that comes with microservices slows down development and can bring much additional complexity[15]. Furthermore, an overhead is introduced when having to manage distributed communication over multiple services.

This thesis aims to provide valuable insights into the trade-offs involved, especially regarding cost efficiency, when scaling a system vertically versus horizontally. To enhance decision-making in the design of scalable software architectures running in the cloud.

The thesis is done in collaboration with the Financial Technology company *Clear Street* and will focus on the typical workflows found in financial industry systems.

1.1 Research Questions

To achieve the objectives of this thesis, the following research questions will be answered:

- **RQ0**: What bottlenecks arise in the monolith and microservice that limit them to effectively scale up?
- **RQ1**: When should a monolithic architecture and vertical scaling be chosen over a microservice architecture scaling horizontally, based on the performance-to-cost function?

1.2 Thesis Outline

This thesis paper is divided into the following sections:

- **Section 2 |** Gives an introduction and theoretical background about the relevant areas and fields which the thesis touches
- **Section 3 |** Introduces previous research in the are, their result and how this thesis builds upon it.
- **Section 4 |** Present the methods and system used to gather the data needed to answer the research questions.
- **Section 5 |** Presents the result of the experiments. This is an iterative process, and discussions are held between each iteration to motivate the direction it is moving in.
- **Section 6 |** Discusses the result of Section 5 and tries to find answers to the research questions.
- **Section 7 |** Concludes the thesis, suggesting possible improvements, potential threats to validity, and suggestions for the direction of future work in the field.

2 Background

The following section will give an introduction and necessary context about the relevant areas and fields which the thesis touches. Starting by introducing distributed systems and cloud computing, we move on to the architectures used in this thesis and how they will be deployed in the cloud. We will then progress to the two scaling approaches and how they will be implemented. Then, some technologies and platforms will be introduced which will be encountered later on. Finally, the section will describe some essential financial terms and some financial systems and their architectures, systems which will be used as inspiration for the system developed and used for this thesis.

2.1 Distributed Systems

A distributed system can loosely be defined as "*a collection of autonomous computing elements that appears to its users as a single coherent system*" [8].

The "collection" characteristic in a distributed system comes from the fact that the system consists of multiple nodes or computers working together. The system nodes do not need to be homogeneous and can vary from high-performance computers to smaller IoT devices [8]. The central principle of distributed systems is that they can operate independently of each other but are often programmed to work toward a common goal [29].

The introduction of a collection of nodes also introduces numerous synchronization and coordination problems. From this, several consistency and fault tolerance issues could arise around the system. An example of such a coordination problem is that each node will have its own view of time, which leads to questions about synchronization in the system. Another example is that a distributed system needs to know if a node is alive, dead, or only temporarily silent, adding additional complexity to the synchronization [29].

Furthermore, a distributed system is often geographically dispersed, which means that it is accessible to a wide range of users. But it also means that it comes with more fault tolerance; if a location goes down, the system can still function as long as it can do the work in the other locations[29].

The other key characteristic "coherent system", means that the distributed system should act as a single coherent system. This means that the system produces the same result, no matter where, when, and how the user interacts with it [29].

2.2 Cloud Computing

The definition of cloud computing by the National Institute of Standards and Technology[1] (NIST) is: "*Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be quickly provisioned and released with minimal management effort or service provider interaction.*" [23].

The main advantage of the cloud is that a user only has to pay for the resources they use.

[1] https://www.nist.gov/about-nist

Unlike owning your own hardware, where you have to pay a higher cost upfront and often over-provision the hardware in case of sudden load increase [6]. Today, cloud computing is often provided by public cloud providers, such as Amazon Web Service[2], Google Cloud[3], and Microsoft Azure[4] [27].

2.2.1 Cloud Native

Cloud native is a term used to describe applications and software developed for the purpose of running on the cloud. This means that the system and its architecture are designed to run in the cloud, as a distributed system, and handle the issues that come with it. Some basic properties to describe cloud native applications are: Operating on a global scale, that is, accessible anywhere on the globe at a minimal latency. Being able to scale quickly when the workload increases. Designed in a way that allows for seamless upgrade and testing of different parts of the application, without disrupting system operations [16].

2.2.2 Amazon Web Service

As mentioned in the Cloud section 2.2, Amazon Web Service[5] or AWS is a cloud provider. It provides several services such as Amazon Elastic Compute Cloud (Amazon EC2), which is a computing platform that offers different types of compute instances. You can choose between Intel, AMD and Arm processors, how many CPU cores and RAM for each instance, network bandwidth, et cetera [1].

Another service is Amazon Elastic Kubernetes Service or EKS, which is a service that allows running Kubernetes clusters in the Amazon cloud (see the Kubernetes section 2.6). The services remove the need to operate and maintain a cluster by yourself and are instead managed by AWS, the cluster is supported by EC2 nodes [35].

AWS also provides database services such as Amazon Relational Database Service or RDS, which enables you to set up, operate, and scale a relational database (Postgres, MySQL, et cetera) on AWS. Amazon RDS can be deployed to EC2 nodes, giving you the ability to choose the resources in more detail, while AWS manages the hardware layers for you [36].

2.3 Software Architectures

A software architecture refers to the collection of structures necessary to grasp/understand a system. It includes software components, the relationships between them, and the attributes of both [24].

2.3.1 Monolith

The monolith is a system architecture consisting of multiple modules with different responsibilities; however, the modules cannot be executed independently. They are deployed as a single application and run as a single process [24].

[2]https://aws.amazon.com
[3]https://cloud.google.com
[4]https://azure.microsoft.com
[5]https://aws.amazon.com/

The monolith has a simple architecture and is often easy to maintain and develop on a smaller scale. Debugging and following the data flow through the system is more straightforward, and there is no communication overhead introduced between modules[21].

However, once the application grows larger and more complex, updating and maintaining it becomes an issue since the modules can have an increasingly high coupling. This, in turn, can make the application lose its modular structure, making it harder to debug and maintain. For example, teams will have a harder time coordinating over modules when developing since editing one module can create unexpected behaviour in another module [21].

Finally, since a monolith runs a single process, scaling it horizontally is very limited. As replicas of the whole monolith have to be created instead of only the individual modules that are bottlenecking the system.

2.3.2 Microservice

The microservice architecture also consists of multiple modules with different responsibilities. However, here they are running as their own processes and communicating via some kind of communication tool (Kafka[6], Rest API[7], gRPC[8], etc.). Each module is developed independently, meaning that different languages and technologies can be used between modules (as long as the communication interface stays the same). Moreover, this means that maintaining and updating the system should be straightforward since each module is independently replaceable and upgradeable[15]. However, in reality, the microservice architecture comes with many problems, such as integration tests being almost impossible. Potentially enormous amounts of communication overhead and an increasingly difficult data flow to debug [32].

Microservices can be seen as a cloud native architecture that builds upon the ease of adding and removing compute resources in the cloud. It is ideal for horizontal scaling (see Section 2.4) by adding more of a specific module when needed.

Finally, throughout this thesis, a microservice will only be able to change its own state. This means that the state can only be changed by contacting the service through its API, and each service's responsibilities are limited to a single section. For example, a single SQL database.

2.4 Scaling

As mentioned earlier, scaling refers to the practice of handling a changing workload by adding or removing excess resources to your system [5]. This can be done both vertically and horizontally.

2.4.1 Vertical Scaling

Vertical scaling or scaling "up" is the process of adding more resources to an existing node or machine, in the form of, for example, CPU cores and RAM. This kind of scaling is limited by the maximum amount of resources in a single node and also the hardware utilization of

[6]https://kafka.apache.org
[7]http://redhat.com/en/topics/api/what-is-a-rest-api
[8]https://grpc.io

monolithic applications (horizontal scaling is also limited by hardware utilization; however, the problem is not as critical) [38].

In this thesis, vertical scaling refers to the act of manually and in discrete steps increasing the resources (CPU & RAM) of a single pod that will run the monolith application.

2.4.2 Horizontal Scaling

Horizontal scaling or scaling “out” is the process of adding entire nodes to the system, where each node often adds the same amount of resources to the system. Horizontal scaling is more complex than vertical scaling and is built on the acquisition of information about individual modules and the decision of what, when, and how much to scale. [38]

In this thesis, horizontal scaling refers to the act of manually increasing the number of pods/instances of a service running in a Kubernetes cluster. This will result in a discrete increase in resources.

2.4.3 Load Balancers

Load balancers are used to distribute workload across multiple instances of a service [31]. They are needed to ensure efficient handling and reduced response time by distributing the load across the entire system. Ensuring that no single node/service is overworked or underworked.

2.5 Kafka

Apache Kafka is a high-throughput distributed event and message streaming platform. It can be used to send and receive messages or “events” through queues or “topics”. A user can either subscribe to a topic to consume events, or the user can be a producer and publish events itself to a specific topic for others to consume. Kafka uses persistent storage, meaning that it will build up an event history or topic log, saved on disk. Ensuring data integrity and reliability, even during server failures [37].

2.5.1 Partitions

Kafka allows its topics to be divided into “partitions”, allowing multiple users to subscribe and consume events simultaneously. If users are in the same consumer ”group”, Kafka ensures that a single event is only consumed once per group [20].

Based on this, Kafka and partitions can be used to ensure that a consumer always has the same “type” of events by choosing which partition to produce on, based on a specific key. For example, by hashing a unique symbol, we can ensure that the same consumer will always handle any events related to that symbol.

2.6 Kubernetes

The official definition for Kubernetes[9] is “*Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative*

[9] https://kubernetes.io/

configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support and tools are widely available." [26].

It functions as an orchestration system for containers. A container is a package containing an application's code and any required libraries and/or dependencies needed to run the application. This enables the application to run anywhere since all its dependencies are packaged into a single file, which allows for easy distribution in cloud environments [34].

Containers are deployed using "pods", which is a group of one or more containers. When a container is deployed via Kubernetes, information such as the number of replicas and resources can be defined. This allows Kubernetes to start a number of pods with a predefined number of CPU cores and RAM, all running the same container(s). It then continuously monitors the deployment, providing auto-restarts and re-scheduling on failures, to ensure that the desired state is held [28].

2.6.1 Istio

Istio is a Kubernetes service mesh. This is a tool that can help with the communication between services in, for example, a microservice architecture. It adds reliability, security, and observability by managing communication between pods. This is done by adding a proxy to each service, which intercepts network traffic to and from the service, meaning that communication is between the proxies rather than between the services directly [33].

2.7 Financial Background

The following section will introduce some financial terms and some typical financial architectures.

2.7.1 Financial Terms

There are several financial terms used throughout this thesis, which are central in achieving the final result [17].

- **Equity** | Represents an ownership in an entity, such as a company. Providing the owner with a claim on future earnings and assets.
- **Option** | A contract that gives the owner the right, but not the obligation, to buy or sell an underlying asset at a specific time (maturity date) for a specified price (strike price) for a cost called a premium. An option can either be a "call" option, giving the buyer the right to sell the underlying asset. Or, it can be a "put" option, giving the buyer the right to buy the underlying asset.

  There are two types of options "European" and "American", the European option can only be exercised at the maturity date, while the American option can be exercised at any time between the purchase and the maturity date.
- **Financial Instrument** | A financial instrument is a monetary contract that results in a financial asset of one type going to one party and a financial liability or a financial asset of another type going to the other party. Two examples of these are Equity and Options.

- **Position** | A position can either be "long" or "short". A long position is created when an individual, via a buy-order, acquires new assets and is closed when the assets are sold. A short position means that an individual sells an asset in the hope of repurchasing it later at a lower price, profiting from the potential decrease in asset price.
- **Profit & Loss (P&L)** | P&L is the profit or loss that an individual or entity has made in their trades. It comes both a realized P&L, meaning profit or loss that has been realized by exiting a position. And as unrealized P&L, the profit or loss that would be made if the individual or entity exited a position at that moment.
- **Exchange** | In finance, an exchange is a place where financial instruments can be traded between traders; it ensures fair and orderly trading. Such as the New York Stock Exchange (NYSE).
- **Broker** | A broker acts as an intermediary between an investor and an exchange. This is necessary as most exchanges only allow orders from individuals or companies that are members of the exchange, meaning a "normal" individual will not be able to trade directly with the exchange.

2.7.2 Financial Systems

A financial system is characterized by high throughput and low latency. Some systems claim to have trade latency down to a few tens of microseconds. Distributed systems are also key to allow high throughput and fault tolerance [12].

As mentioned in 2.7.1, an exchange facilitates the trading of financial instruments between individuals or entities. This is done by holding a matching book for each instrument that can be traded. The matching book holds the depth of each price bid, meaning the total number of sell-orders or buy-orders at specific prices, from the current active trades. The main objective of the exchange is to match the sell and buy orders by price and time, which is carried out by a matching engine [10].

More interesting for this thesis is the stock broker. The broker handles orders coming in from their members; this means things such as controlling the validity of the order and calculating the risk of the member, that is, if the member has the financial power to complete the order. Stock brokers can also offer features such as summaries of a member's trading history and performance to allow members to gain insight into their trading performance. In the end, the broker has to place the order on an exchange to complete it [30].

Finally, financial systems such as exchanges and brokers are often developed to adhere to some kind of event-driven architecture [10]. Event-driven architecture (EDA) is an architecture that uses events to communicate between services, for example, an order being placed at a broker or an underlying instrument needing validation (Control if it exists) [11].

3 Related Work

The discussion about the monolithic single application architecture and distributed microservices has been ongoing since microservices came on the scene in the 2010s. However, studies which try to find an answer to microservice scalability versus monoliths are still rare, and more work needs to be done.

However, some research has been done to answer the questions about scalability and performance between monoliths and microservices. Bjorndal et al. [3] established that even though the monolith can outperform the microservice in cloud environments. The cost to performance could be up to 6 times higher for the monolith when scaling both systems. Coming to the conclusion that the monolith performs better for small to medium-sized systems, but when the workload increased to a large number of concurrent users, the microservice architecture became cheaper to scale.

Blinowski et al. [4] conclude that there was a visible cap on horizontal scaling, where scaling could worsen the performance. The study also showed that vertical scaling became limited by the most powerful virtual machine available.

This thesis aims to build upon these two studies by not only answering when one architecture and scaling approach is more favourable, but also looking into what limits the architecture's scalability and performance. Identifying which properties to look for when deciding on the system architecture.

Furthermore, both studies mention that it would be interesting and improve the validity of the result to use a system and a dataset that more closely resemble real-life systems used by companies. This means making the systems bigger and more complex but also using a larger and more complex data set that resembles ”real” data. Both studies used relatively simple and non-complex systems where communication between microservices was quite limited. But they also mentioned adding security to the systems to make them more resemble real-life systems.

This will be achieved by developing a system that more closely resembles the real systems used by companies. Through the collaboration with the financial technology company Clear Street, a more complex system with more realistic service communication will be developed, which also takes into account things such as communication security. Furthermore, the data sets used will be "real" test data, in the form of the internal test data used by Clear Street to verify the performance and functionality of their system. This consists of a combination of real production data and generated data.

4 Method

This chapter will introduce the method, systems and architectures used to collect data and answer the research questions (**RQ0** and **RQ1**). Starting with a higher system overview, showcasing the dataflow. Introducing the data sets that the systems will use, how these data sets are loaded (and generated) into the systems, and how the systems store the information created during the dataflows.

Then, both the monolithic and the microservice system designs will be introduced, including, amongst other things, sequence diagrams. After the design of the systems is introduced, the infrastructure and environment in which the systems will run are described. Finally, the measurements that will be used to collect the results are introduced.

4.1 System Overview

An important basis of the thesis is that the systems used when scaling and running the experiments need to more closely resemble the real systems used in the industry. To achieve this, a "simplified" brokerage system is used which is then scaled vertically and horizontally by implementing it as both a monolith and a microservice.

The system consumes incoming orders and validates them against available instruments and accounts. If not successful, it will ignore the order and produce an appropriate error message. If successful, the corresponding position will be updated and the order and any realized P&L will be saved. In parallel, it will consume and save changes in the price of the equity instrument and recalculate the unrealized P&L of affected positions.

As can be seen in Figure 1. The system will have two workflows. The first workflow is the "Order Consumer" which consumes an order from the Kafka topic and goes through the following steps:

1. Consume a new order.
2. Validation (see Data Retrieve/Validation steps in Figure 1.)
   (a) Control that the account for which the order is performed exists and is valid.
   (b) Control that the instrument that the order tries to trade exists, if option control that the maturity date and that the option is valid.
   (c) Control if the member that owns the account can go through with the order (financially) and fetch the price of the instrument; if it is an option, the price will be calculated based on the underlying instrument.
3. State Update (see Data State Change steps after validation in Figure 1.)
   (a) Update affected position or create a new one.
   (b) Save the order to the database.
   (c) Update realized P&L, if any position was exited.

If we keep looking at figure 1. We can also see the second data flow, which is the "Price Update Consumer" that consumes incoming price updates. It has the following steps:

1. Consumes a new equity price.
2. Update the corresponding instrument in the internal price table. (See In-memory store in Figure 1.)
3. Update unrealized P&L for each position holding the updated equity and all positions containing options, whose underlying instrument is the updated equity.
   - The unrealized P&L is not updated instantly for each incoming price update, instead, the consumer will collect several updates during a predefined period. The unrealized P&L will then be updated for all instruments that had a price update during that period. If the consumer falls behind when updating the P&L, it will jump over any price updates from the same instruments, moving to the latest price update for each instrument.

The load on the system will be generated by a supplementary application, called "Load-Generator" in Figure 1. It has the responsibility to send orders and price updates, but also pre-generating member and accounts based on the orders at hand. The orders will be prefilled into the Kafka topic, allowing the system to consume the orders as rapidly as possible.

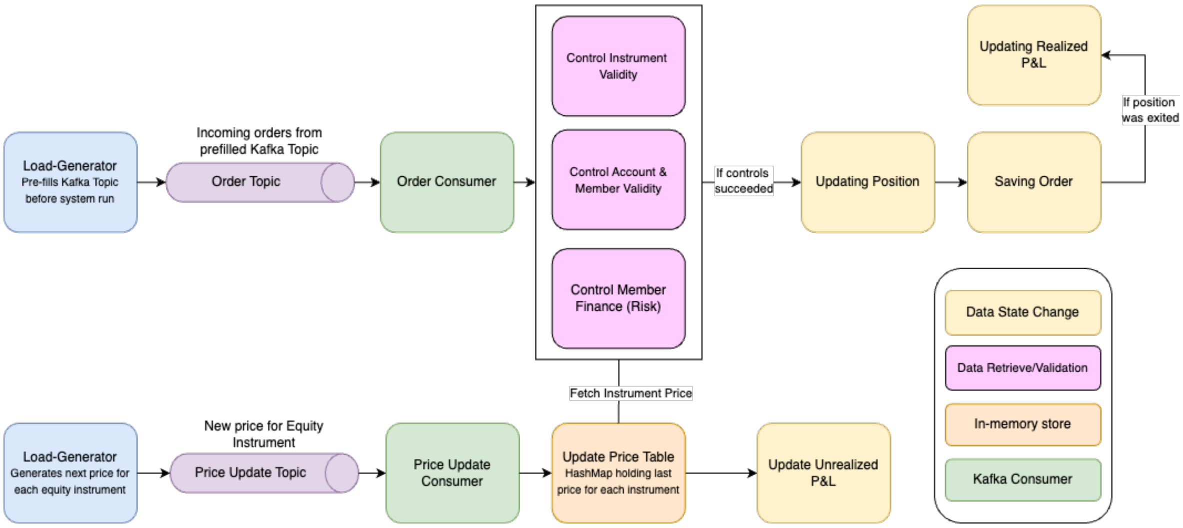

Figure 1: System overview, order and price update dataflows

4.2 Datasets

The system uses a total of four datasets. An instrument data set that contains the instruments that can be traded in the system. An order data set that contains all the orders that are being placed. A price data set that holds the start price for the instruments. And a final data set that contains the price updates; however, this data set is generated during each run based on the start prices.

4.2.1 Instruments

The first data set is a collection of around 800.000 instruments, consisting of two types of instruments "Equities" and "Options". All orders will be placed for instruments that reside in this data set. The *Option* columns are only populated if the instrument is an Option. The instrument column headers are the following:

- *Instrument Id*
- *Currency Code*
- *Exchange*
- *Instrument Name*
- *Instrument Description*
- *Instrument Type(Option, Equity)*
- *Option Underlying Instrument Symbol*
- *Option Underlying Instrument Id*
- *Option Maturity Date*
- *Option Type(Put, Call)*
- *Option Strike Price*
- *Option Strike Multiplier*
- *Option Exercise Style*

4.2.2 Orders

The second data set that is used is the orders being placed; there are just under 100 million orders in the data set. This data set contains 66 columns; however, most of them are trade-specific and can be ignored. As they mostly only provide information to Clear Street's internal systems. The most important are the following:

- *Order Id*
- *Account Id*
- *Instrument Id*
- *Currency Id*
- *Price*
- *Quantity*
- *Side (Sell, Buy)*
- *Short (If short position)*
- *Trade Date*

4.2.3 Start Prices

The third data set provides the start prices and volatility of all equity instruments in the first data set. Each instrument start price has been calculated from multiple closing prices (multiple days).

These values are loaded and used to generate a stream of price updates; see 4.2.4 for how price updates are generated. Important to note is that option prices are calculated when needed, based on the underlying instrument price. The columns in this data set are as follows:

- *Instrument Id*
- *Volatility*
- *Price*

4.2.4 Price Updates

The last data set is the price updates. A price update is used to simulate other trades and orders placed on the instrument, moving the instrument price. These will be generated during the system runtime and will be sent at a pre-defined rate to a Kafka topic. From which the system can consume the price updates continuously.

This is done since the run time of each experiment is unknown beforehand. The run time depends on throughput, which is unknown before the system has been tested. That is, the exact amount of updates needed is also unknown before running. Therefore, it is easier to generate price updates at run time. However, this leads to some inconsistencies, as every run will use a different update series, leading to different P&L results at the end of each run.

Each price update has a chance to be cancelled, that is, the price update will be skipped. This means that different instruments will have different amounts of incoming price updates over the run-time. This is done to simulate the popularity difference amongst traders between different instruments. The chance will be skewed based on the instrument's popularity amongst the order dataset, and the instrument identification will be used as a seed to ensure consistency over multiple runs. This means that each instrument will have a unique "update rate".

The generation of the prices is done using a Geometric Brownian Motion (GBM)[10]; more exactly, a one-dimensional GBM called the Wiener Process is used. GBM comes from physics and describes the random movement of particles in a substance. This can then also be used to simulate intra-day asset price movements based on a predefined drift or volatility. The formula for this process can be seen in Figure 2.

[10]https://en.wikipedia.org/wiki/Geometric_Brownian_motion

$$r = \mathcal{N}(0,\, \delta^2) \tag{1}$$
$$w = \sigma * r \tag{2}$$
$$gmb = (w + (\mu - (\sigma^2/2)) * \delta) \tag{3}$$
$$nP = p * gmb \tag{4}$$

where:
δ **is the time step (seconds per update)**
μ **is the overall price drift (trend of prices)**
p **is the starting price**
nP **is the new price**

Figure 2: Wiener Process, used to generate price updates in instruments.

The starting price and volatility from the third dataset are used as parameters for the first iteration, and the newest generated price is then used as the next start price in the next iteration. The overall trend (mu) is set to zero, meaning the prices will not tend to drift towards any direction more than another.

4.3 Storage

A *PostgreSQL*[18] database is used as storage. The columns in the tables are mostly equal to the data columns described in the Dataset section(4.2). The following tables exist:

- **accounts**
  - A prefilled table, based on the orders that have been filled into the topic, and is therefore only read from (no writes).
- **members**
  - A prefilled table, based on the orders that have been filled into the topic, and is therefore only read from (no writes).
- **orders**
  - The order table is filed with each incoming order and will be read continuously for different calculations.
- **positions**
  - Positions are created and edited based on incoming orders. Each position entry is also holding an unrealized P&L value, which is calculated based on the incoming price updates.
- **realized_pnl**
  - Realized P&L rows are created continuously as positions are exited and also continuously read from during different calculations. They are mapped to the order that exited the position.
- **instruments**

- The instruments table is prefilled with the valid instruments and their information and is therefore only read from (no writes).

4.3.1 SQL Optimization

When updating the unrealized P&L, the database has to find and update multiple positions. Many of them are options whose underlying instrument has received a price update; to find these, a lot of matching between tables is required. To speed this operation up, the *Underlying Instrument Identification* column in the *instrument* table is indexed. Furthermore, to speed up the updating of the affected positions the *Instrument Identification* column in the *position* table is also indexed.

4.4 Monolith Design

To get a fair comparison between the monolith and the microservice, the monolith will be parallelized by partitioning. This will be done by running multiple Kafka consumers; see the Resource section 4.9 for more details. This means that the monolith will consist of x number of threads that consume orders (see order dataflow in Figure 1). And y number of threads that consume price updates (see price update dataflow in Figure 1).

The monolithic design allows the system to perform a lot of calculations and fetching by using more complex SQL queries, which puts more of the CPU usage on the more optimized PostgreSQL engine and removes unnecessary fetching from the database. For example, updating positions and aggregating P&L over a member can be done in a single SQL query.

4.4.1 Order Processing

As shown in Figure 3. The order module consists of multiple order consumer threads; these are the “Kafka Consumers” that can be seen in the order data flow of Figure 1. Each thread is a Kafka consumer and has its own “Instrument”, “Account” and “Order” modules. Since each module uses its own database connection, a “consumer thread” can perform parallel SQL queries from its internal modules.

The instrument and account/member validation modules in Figure 1 are performed using these modules, while the final finance/risk control is done using a larger SQL query, which aggregates unrealized and realized P&L over a specific member. If any of these validity calls fail, the order is not processed/handled.

To further increase the performance of the monolith, these three “validation modules” will be performed by Java Virtual Threads[11]. This is done since the modules consist mainly of calls to the database, meaning that the majority of the time will be spent waiting on IO. Performing the modules in parallel will therefore have a low impact on the performance but will increase the throughput of the system.

[11]https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html

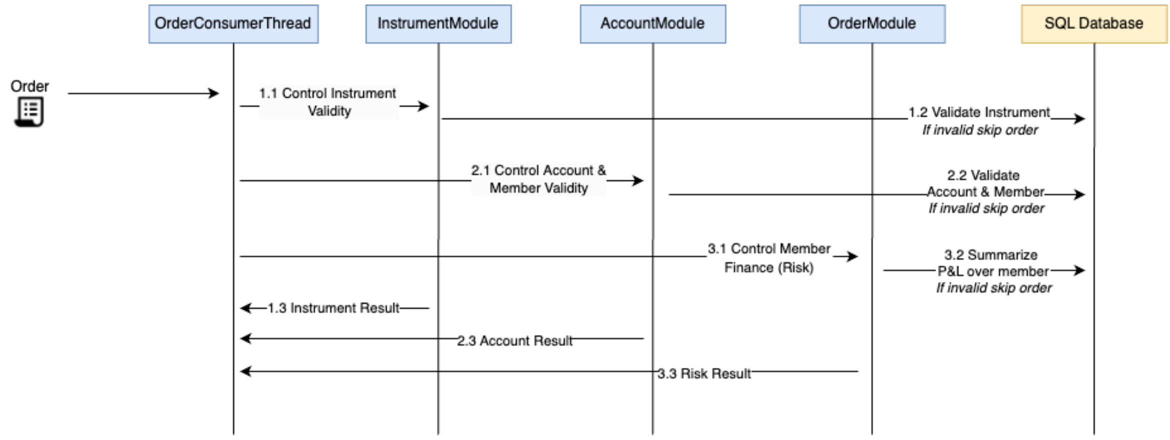

Figure 3: Order validation at Monolith (Sequence Diagram)

If all three modules are successful, the thread will fetch a price for the order and then send the order to its "OrderModule", as shown in Figure 4. Which will update the position; this operation can fail if, for example, an order tries to sell assets the account does not own. It will then save the order to the database and update the realized P&L if needed.

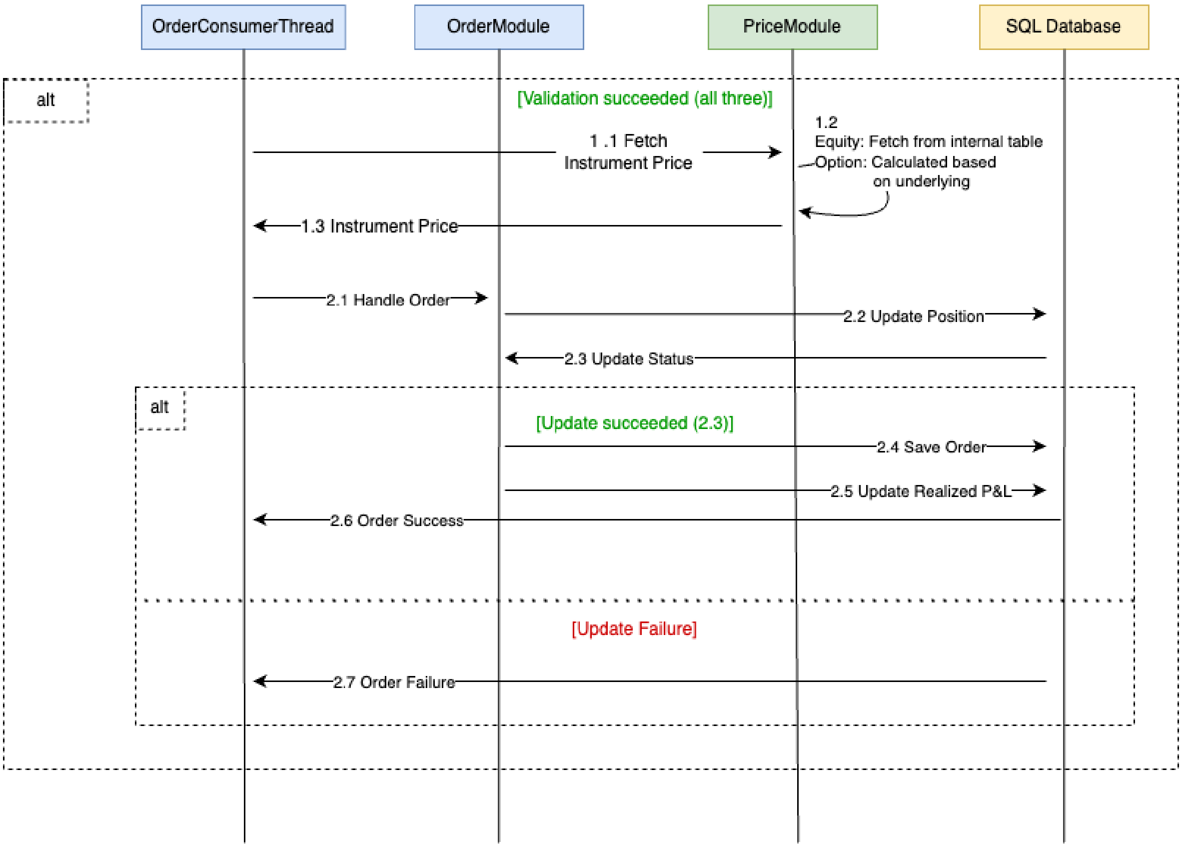

Figure 4: Order processing at Monolith (Sequence Diagram)

4.4.2 Price Processing

The price module is built upon the same design as the order module, with multiple price consume threads, see Figure 5. Each thread starts an additional internal thread, in which a Kafka consumer handles the incoming price updates, where it updates a shared *HashMap*[12] called “PriceTable”, which saves a value at a specific position based on hashing a key. This “PriceTable” is then accessed by the ”broker threads”, to fetch instrument prices.

The price module also has the responsibility to update any unrealized P&L. As mentioned in Section 4.1, the unrealized P&L is not updated for each incoming price update. Instead, after the price thread has created the internal consumer thread, it goes into a loop, in which it wakes up and recalculates the unrealized P&L at a predefined interval. A thread will only update the P&L for the instruments that come in on the partitions that it has been given.

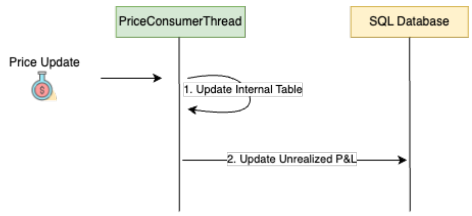

Figure 5: Price update processing at Monolith (Sequence Diagram)

4.5 Microservice Design

The microservice consists of a number of independent services, each having access to a single SQL table (some have access to none), in order to ensure consistency through a single writer. The services will communicate with each other using gRPC and each service instance uses a PostgreSQL connection pool to communicate with the database. The connection pool will allow each service to handle multiple requests in parallel; the size of the pool will be decided later.

Whereas the monolith performs alot of calculations and fetching using more complex SQL quires, the microservice will have to resort to simpler SQL quires. Since each microservice only has access to a at most a single SQL table, more complex quires that require multiple tables are not possible. Instead, fetching of the needed data will have to be done by multiple services, aggregated at one service, where possible calculations can be performed.

4.5.1 Order Processing

Looking at Figure 6. The order component of the microservice works similarly to the monolith. An order consumer service will act as a consumer, consume an order, and then validate it. This will be done by contacting the *InstrumentService* and *AccountService*, which validates the instrument and the account, it also contacts the *PnLService* to control the member risk. This service/module does not exist in the monolith and holds responsibility for any P&L calculation which needs to be performed. This is done by contacting a number of different services to retrieve all needed information and aggregating a result.

[12] https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

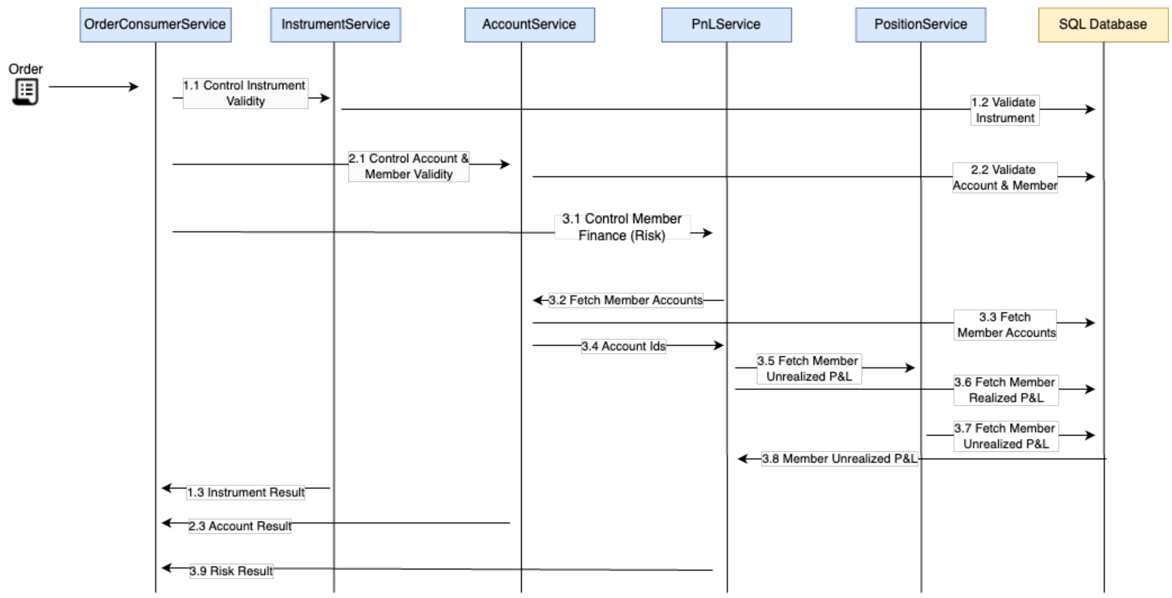

Figure 6: Order validation at Microservice (Sequence Diagram)

If all validations are successful, the order consumer service proceeds to the sequence seen in Figure 7. First, the consumer contacts the *PriceFetchService* (which will act as a proxy), and it is responsible to contact the correct *PriceService* instance. This is needed as the previous ”PriceTable” will now be split over multiple *PriceService* instances, as the *PriceService* is partitioned over Kafka partitions.

To solve this a *PriceService* will inform the *PriceFetchService* of its instrument range when starting, giving the *PriceFetchService* knowledge of where to fetch each instrument price. After obtaining the price, the consumer will send the order to the *OrderService*. This will take care of updating the affected position, saving the order, and updating any realized P&L. As can be seen in Figure 7, this is done by contacting multiple services.

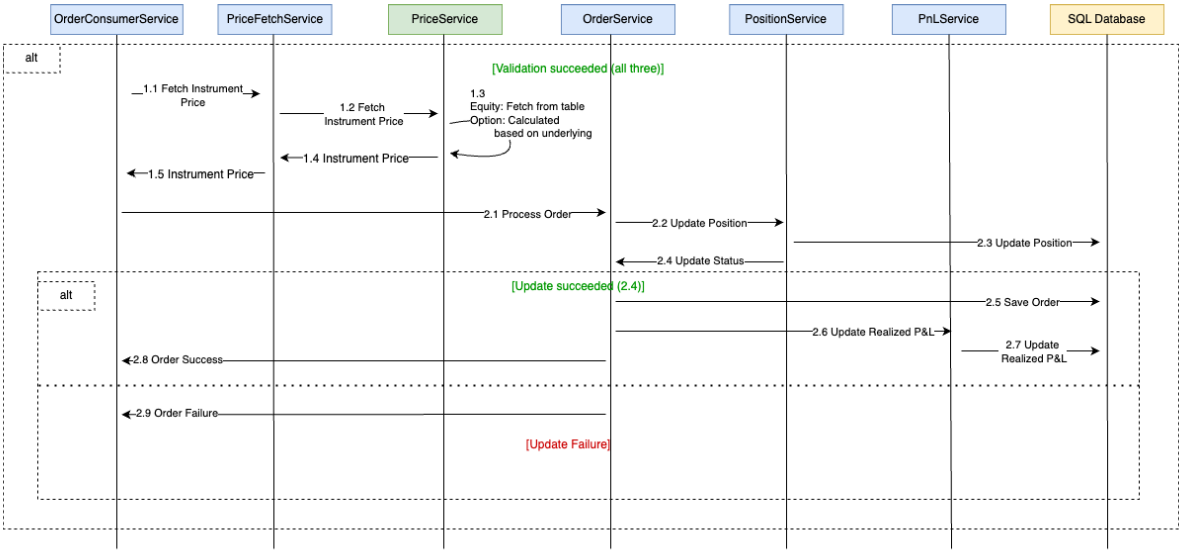

Figure 7: Order processing at Microservice (Sequence Diagram)

4.5.2 Price Processing

The price update sequence for the microserivce can be seen in Figure 8. Here multiple *PriceServices* acts as consumers, updating the internal price table as new price updates come in. Each price service also has the responsibility to update any unrealized P&L. As mentioned in Section 4.1, the unrealized P&L is not updated for each incoming price update. Instead, a local internal scheduler tells each service at a predefined interval to update the unrealized P&L of the instruments for which it has collected a price update for the latest period.

The unrealized P&l for an instrument is updated by contacting *PnLService*. The *PnLService* in turn contacts the *InstrumentService* to check if there are any options that use the instrument as their underlying instrument. If there is, the *PnLService* will contact the *PriceFetchService* to fetch the option prices. This in turn contacts the correct *PriceServices*, which calculates the option prices based on the underlying instrument price. Finally the *PnLService* will contact the *PositionService* to update the affected positions of the instrument and any options.

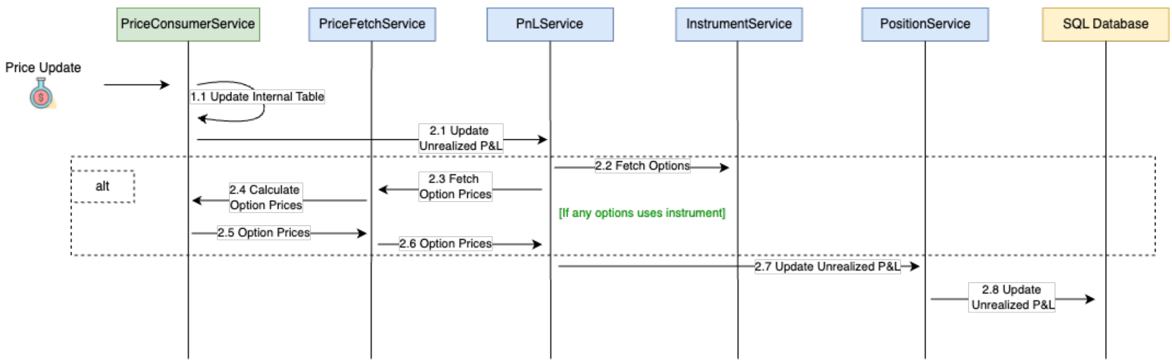

Figure 8: Price update processing at Microservice (Sequence Diagram)

4.6 Infrastructure

Both the monolith and the microservice will be deployed in a Kubernetes cluster running on top of a signle EC2 node at a time. This is chosen instead of running only the microservice in Kubernetes and the monolith directly on an EC2 instance. As it is easier to ensure that both architectures have the same amount of resources if both are running as pods.

The monolith will be deployed as one large Kubernetes pod, while the microservice will consist of multiple smaller pods, running multiple replicas of each service. The total amount of resources used by the microservice pods will be equivalent to the monolith pod. To see specific resources for both systems, see Section 4.9. Moreover, both architectures will run on the same type and number of nodes during each measurement; see Section 4.9.1 for node details. This is needed to accurately calculate the cost of the system, as the cost will be the hourly rate for the EC2 nodes. That is, to make the comparison fair, each architecture has to fully utilize the available nodes.

On top of the Kubernetes cluster, the Istio service mesh will be run. The reasons for deploying pods using a service mesh are; Service mesh such as Istio provides mTLS encryption between services/pods [22]. This was a priority, as both [3] [4] from Related Works (Section 3), mentioned adding security as an improvement to their papers. Secondly, since the microservices will communicate using gRPC and gRPC uses HTTP/2 connection. The Kubernetes default load balancing will not work as intended, since Kubernetes load balancing works by assigning each new TCP connection to an address. However, since HTTP/2 uses a single long-lived TCP connection, each gRPC request from a client would go to the same address. Istio solves this by setting up TCP connections to each address and then load-balancing each outgoing call over these TCP connections instead [19].

Furthermore, Istio provides an extensive set of tooling for monitoring the external behavior of our pods, this includes HTTP metrics (request volume, success rate, and latency distributions), TCP metrics, and more. This provides a way to monitor communication and IO between services, resulting in another tool when profiling the systems.

Although most of the advantages for Istio come into play when running the microservice, it will also be used for the monolith. As it provides a way to profile the networking when communicating with the database.

4.6.1 Storage Setup

The database will also be deployed using Amazon EC2, running on its own on an R5b[13] node (still part of the cluster). The R5b node type is used since it is designed to run larger SQL deployments. Running it one its own node is wanted since the resources used to operate the database will be included in the cost-metric. The monolith for example can utilize more complex SQL queries than the microservice and therefore could benefit more from a database with higher computational power. Running it on its own node will also enable the database to scale later on, if potential bottlenecks are found regarding the database connection, by increasing the R5b instance size.

[13]https://aws.amazon.com/ec2/instance-types/r5/

4.6.2 Kafka Setup

Kafka is used to produce work for the systems, both in terms of orders and price updates. To minimize the complexity of the comparisons that will be made, the Kafka cluster will not be part of the comparison and, therefore, will not contribute to the cost of the systems or any system performance.

Therefore, the Kafka cluster does not need to run on specific EC2 nodes. As long as the Kafka cluster is given enough resources to not bottleneck the rest of the system during run-time, it will be enough. Furthermore, the Kafka cluster will also not have to scale. which further simplifies the deployment.

Based on these assumptions, some tests and discussion with the Clear Street infrastructure team, a single Kafka broker pod will be used, which will have the resource specifications shown in table 1.

	CPU	**Memory**
Request	4000m	32GiB
Limit	4000m	32GiB

Table 1: Kafka Resource Specification

Two topics will be used, the order topic and the price update topic. The order topic will have 1024 partitions; this is chosen because the number of Kafka consumers will vary between measurements, and with 1024 partitions, all measurement steps will be able to consume from the same topic in the correct manner. Meanwhile, the price update topic will have 128 partitions, using the same argument as for the order topic.

4.6.3 Istio Setup

The measurements is performed on Clear Street's pre-configured Kubernets cluster, which means, amongst other things, that the Istio service mesh is also pre-configured. In order for the systems to use Istio, each service instance/pod will be deployed with a specific label. This inserts an Istio sidecar as a container into the pod, allowing load balancing for gRPC and mTLS between pods.

4.7 System Configurations

In order to ensure a fair comparison between both systems when answering **RQ1**, both the monolith and the microservice should run in a somewhat optimal configuration. That is, the way the systems/architectures utilize the given resources is configured to provide the best performance. It is not feasible to find the most optimal solutions in this thesis, as that would take too long; however, investigating what an *more* optimal solution for both architectures would be (compared to the worse case) is enough to ensure that one of them will not be significantly worse due to their resource setup.

To answer this question, an investigation into both systems has to be done; however, this investigation would in essence be an investigation of possible bottlenecks of the systems, which would be part of *RQ0*. Therefore, the following sections will only present the configuration which both architectures will initially use when answering *RQ1*. The road that leads to these configurations and any arguments made for them can instead be found in the Result & Discussion Section 5, where it will be part of the discussion regarding *RQ0*.

4.7.1 Monolith

For the monolith, the essential configuration is the number of order consumer threads and the number of price consumer threads, per the number of virtual CPU cores available at the underlying node. The monolith is configured to run two order consumer threads per virtual CPU core and a one price consumer per seven vCPUs (this will result in one price thread being added at each "scale step"). Furthermore, the monolith has all the CPU and memory available at the underlying node. See Section 5.1.1 for the discussion that came to this conclusion.

4.7.2 Microservice

The microservice configuration used in the first resource step can be seen in Tables 2 and 3. Additionally, each order consumer pod also utilizes 32 virtual threads when processing the orders; there is still only one Kafka consumer, but the work is divided over virtual threads to increase parallelization. The message ordering per partition is still intact, as all messages from a single partition will always go to the same virtual thread.

To effectively handle this increased parallelization, every service that communicates with the database now uses a "connection pool", with a maximum of 32 connections per service. This will allow, for example, a single instrument service to handle 32 database calls in parallel (if there are enough resources to use). See Section 5.1.2 for the complete procedure and argumentation to get to the final configuration.

The additional resource steps all use the initial configuration as their "blueprint", but are scaled according to the additional resources. See Section 4.9 for how the microservice is scaled and the instance count at each step.

Table 2: Pod instance count used in the microservice at the first scaling step

Service	Pod Count
OrderConsumer	4
PriceConsumer(Service)	4
OrderService	1
PositionService	1
PnlService	1
AccountService	1
InstrumentService	1
PriceFetchService	1

Table 3: Resource for pods used in the microservice at the first scaling step

Service	**CPU**	**Memory**
PriceConsumer(Service)	500m	1800MiB
All other services	360m	1800MiB

4.8 Measurement Definition

A **measurement** is the act of measuring the throughput of both architectures, with varying computing resources. To accomplish this, an "experiment" is defined as when either the monolith or the microservice handles all orders in the pre-filled order topic (4.2.4) as fast as possible. During the same time, the system consumes price updates and updates the unrealized P&L for any affected positions. This results in an order throughput and a P&L calculation throughput, representing the two data flows in the system.

A P&L calculation is defined as the update of all positions of the instrument that have an incoming price update; this includes any positions of options that use the instrument as their underlying instrument.

The main objective of the experiments is to fit the performance-to-cost functions from **RQ1**, for both the monolith and the microservice. The performance metric is defined as our throughput and the cost metric is defined as the hourly rate of the current node set (see Sections 4.6 and 4.9). This includes the cost of system resources and also the database resources used to perform the experiment.

To fit these functions, the experiments need to run with varying compute resources. This is accomplished by manually scaling the resources for both architectures, in discrete steps, by running the systems on increasingly larger EC2 nodes (see Section 4.9). Each equivalent step between the architectures uses the same total amount of resources. In the end, these functions are used to determine when one architecture is scaling more effectively than the other, meaning that the most performance gain is obtained for the least amount of cost.

As mentioned in section 4.6.2, the resources used to operate Kafka is not included in the metric cost, and no scaling of the Kafka cluster is done. The Kafka cluster will be deployed at the beginning, with the resources needed to handle the maximum load of the monolith and the microservice in the last scaling step (see Section 4.9).

Besides fitting the performance-to-cost functions, this thesis also aims to find any bottlenecks that reside within the implementation while scaling them. This is done by numerous profilers when running the experiments, to collect information about: CPU cycles, CPU time, wall-clock time, method calls, SQL latency and queries, network data, et cetera. More information on the specific profiler and how they are used can be found in Section 4.8.2.

4.8.1 System Load

The system load for all experiments/measurements is the same, which means that all experiments are handling the same number of orders; the number of orders to handle will be *1,000,000*. There are also *10* "warm-up" orders per partition, totalling 10240 orders with the current Kafka setup, the warm-up orders are used to avoid any cold start of the components/services.

As mentioned in Section 4.2.4, there is a predefined rate at which the price updates are produced to Kafka. This rate will be constant throughout all experiments. To decrease the overall complexity, the rate is defined as 5000 milliseconds.

Furthermore, the topic partition to which a price update is being sent is based on the instrument identification, meaning that all updates for an instrument go to the same partition. This is the same for orders, where the partition is chosen based on the instrument identification in the order. With this, some assumptions can be made, such as all positions for a specific instrument being handled by a single consumer. Using this assumption, we can avoid certain race conditions, as the positions affected by incoming orders will always be handled by the same order consumer.

4.8.2 Profilers

To profile the systems, the cloud monitoring service Datadog[14] is used. This is chosen because it already runs inside the Clear Street Kubernetes cluster in which the measurements are performed, and it provides an easy way to gather data from all services. It is used to profile the following types of data:

- CPU Profiling, which collects CPU cycles, stack trace samples of Java methods, native calls to JVM code, et cetera.
- Wall Clock Profiling, collecting wall clock time for method calls, and more.
- IO information, such as TCP latency and more.

To obtain query statistics from the PostgreSQL database, *pg_stat_statments* is used. This is a pre-existing extension included in the *contrib* module and can be enabled to track statistics such as execution time, number of query calls, row modification, locks, and much more. It is used to further investigate whether the database is causing bottlenecks and, if so, how these occur [13].

Finally, Istio's internal monitoring is used to profile the network when services are communicating with each other and when both systems are communicating with the database. The Istio data are available via Datadog.

4.9 Resources When Scaling

The following sections present the resources used by both systems for the scaling experiments, how they are divided over the systems, and the number of service instances.

As mentioned earlier, both architectures run as pods and use the same amount of total CPU and RAM. That is, the sum of all the resources used by the microservices is the same as the resources used by the monolith. Furthermore, both architectures are running on the same types of nodes. As different nodes have different prices, which would have an impact on the cost metric.

Important to note is that there are several agents and "extra" pods running on each node, such as the Datadog agent, networking, et cetera. To avoid the monolith and microservice from being evicted, if there are more requests, then there are resources. Each node is missing

[14]https://www.datadoghq.com/

1 vCPU and 4GiB RAM, compared to what they have according to the AWS specification, this decrease is accounted for in all tables in this section.

Finally, since Istio is used, a relatively expensive sidecar container is launched inside each pod; This container requests *100m* CPU and *128MiB* Memory. This does not significantly affect performance when running the monolith, as there is only one pod. However, when running the microservices, the Istio containers in each service instance quickly add up to a costly amount.

4.9.1 Underlying System Nodes

Table 4 shows the underlying nodes on which both systems can be run. There are four nodes, providing four "resource steps" when scaling, each increasing the available resources for the systems.

Table 4: Underlying System Nodes, which the monolith and microservice will run on

#	Node Type	vCPU	RAM	Hourly Rate (Cost)
S1	m5.2xlarge	7	28 GiB	0.384$
S2	m5.4xlarge	15	60 GiB	0.768$
S3	m5.8xlarge	31	124 GiB	1.536$
S4	m5.16xlarge	63	252 GiB	4.608$

4.9.2 Database Nodes

Table 5 shows the nodes available for the PostgreSQL database to run on. Initially, *r5b.2xlarge* (**D1**) will be used; however, if one of the systems bottlenecks the database, a larger database node will be used for the rest of the steps for that specific system. This means that the systems can run on different database nodes to avoid bottlenecks.

See Appendix A for the different PostgreSQL configuration files used for each node.

Table 5: Available database nodes, that is, nodes on which the PostgreSQL database can be run

#	Node Type	vCPU	RAM	Hourly Rate (Cost)
D1	r5b.2xlarge	7	60 GiB	0.596$
D2	r5b.4xlarge	15	124 GiB	1.192$
D3	r5b.8xlarge	31	252 GiB	2.384$

4.9.3 Monolith Resource Usage

The monolith uses the configuration given in Section 4.7.1, where an order consumer thread per core factor of 2 is used at each resource step. Furthermore, a price consumer is added at each resource step. The monolith runs as a single pod that has all vCPUs and memory at each node.

4.9.4 Microservice Resource Usage

The microservice uses the configuration given in Section 4.7.2, as the configuration for node 1. This configuration is then scaled to run the larger nodes; see Tables 6 and 7 for configurations at each step. Note that the node CPU resource steps are the power of 2 minus 1; in reality, the nodes double their resources, 8, 16, 32, 64. However, each step lacks one vCPU and 4GiB of memory (from the essential pods running on each node). To accommodate for this, the pod resources are increasing a tiny amount for each resource step, instead of adding more pods (more than the double that we add to scale horizontally). Since adding more pods would mean that Istio would essentially take this extra core altogether.

Note that Configuration 4 does not follow the same instance count pattern as the rest of the configurations. This is because the nodes have a limit of 110 pods per node and eight “extra” pods run on every node (agents, et cetera). This means that it can run only 102 pods per node. To accommodate for this limit, two price and order consumer pods have been removed, and their resources have been split over the remaining consumers. Furthermore, the rest of the services have lost one instance each, with the resources also split over the rest of the instances. Each service will have the same amount of resources as if it had scaled as intended

Table 6: Instance count for the microservice configurations used during the scaling

Service	Config 1	Config 2	Config 3	Config 4
OrderConsumer	4	8	16	30
PriceConsumer(Service)	4	8	16	30
OrderService	1	2	4	7
PositionService	1	2	4	7
PnlService	1	2	4	7
AccountService	1	2	4	7
InstrumentService	1	2	4	7
PriceFetchService	1	2	4	7

Table 7: Resources for pods used by the microservice configurations during the scaling

Configuration	Service	CPU	Memory
Configuration 1	PriceConsumer(Service)	500m	1800MiB
	All other services	360m	1800MiB
Configuration 2	PriceConsumer(Service)	550m	2000MiB
	All other services	390m	2000MiB
Configuration 3	PriceConsumer(Service)	560m	2050MiB
	All other services	410m	2050MiB
Configuration 4	PriceConsumer(Service)	600m	2100MiB
	OrderConsumer	450m	2100MiB
	All other services	490m	2100MiB

5 Result

This section will provide the result needed to answer **RQ0** and **RQ1**. The process used when answering **RQ0** is iterative, building on itself to find better configurations and revealing potential bottlenecks.

5.1 Discovering Bottlenecks(RQ0)

When answering **RQ0**, the investigation could be divided into two distinct phases. Firstly, what kind of bottlenecks are directly visible and limiting the performance and how could these be solved to achieve better optimized systems and, with that, better performance (optimizing the systems). And secondly, what bottlenecks arise when the systems start to scale up? (bottlenecks from scaling)

The first phase is also crucial to give both systems a fair starting point when answering **RQ1**, as both systems have to operate in a near optimal configuration to make sure the **RQ1** result is fair. Therefore, the findings and results of this phase will be used to configure the systems when performing the measurements used to answer **RQ1**.

The initial tests to obtain results for the first phase are all performed on the *m5.2xlarge* node (System node **S1**), which has 7 virtual CPU cores (vCPU) and 28 GiB of memory and database node **D1**, see Section 4.9. During the investigation, we will be interested in the two dataflows defined in Section 4.8. The order throughput and the unrealized P&L throughput.

This means that there are two measurements that we want to optimize the systems/architectures around, which increases the complexity. To keep complexity manageable, we will use the premise that both measurements are equally important when tuning the systems for the best performance.

5.1.1 Performance Tuning of the Monolith

In the following section we will try to tune the monolith to a near-optimal configuration. When doing this, the initial aspects in which we are interested are the number of order threads X and the number of price threads Y. From this, the following steps will be performed and their result investigated in order to find the near optimal configuration;

- *Investigating the number of price threads.*
- *Investigating the number of order threads.*
- *Result & Discussion of Monolithic Performance Tuning.*

Effect of Number of Price Threads | *Performance Tuning of the Monolith*
When tuning the monolith for the highest throughput, the initial aspects in which we are interested are the number of order threads X and the number of price threads Y. Initial test runs showed that the majority of CPU time was used to calculate option prices and that order threads were mainly limited by IO, coming from different SQL calls. That is, the CPU will mainly be used by the price threads, with the order threads mainly sleeping, waiting for the I/O operations. Based on this, we decided the first investigation to be how many price threads the monolith can run simultaneously before they reduce each others performance.

Test runs were performed, during which there were 7 order threads, based on the fact that there were 7 vCPUs on the underlying node used. During these test runs, the number of price-consumer threads was adjusted between 1, 3 and 7, having 10 runs each. To determine at which point the CPU-heavy price threads started to negatively affect the order throughput of the system.

From these tests, both order and P&L throughput is the highest with a single price thread, as can be seen in Figure 9. Furthermore, monolith CPU utilization is the lowest with a single price thread, although a single price thread gave the best performance in both order and P&L throughput, as seen in Table 8. Adding additional price threads increases CPU utilization but only worsens performance, indicating that the added price threads introduce context switches that negatively affect both the other price threads but also the order threads.

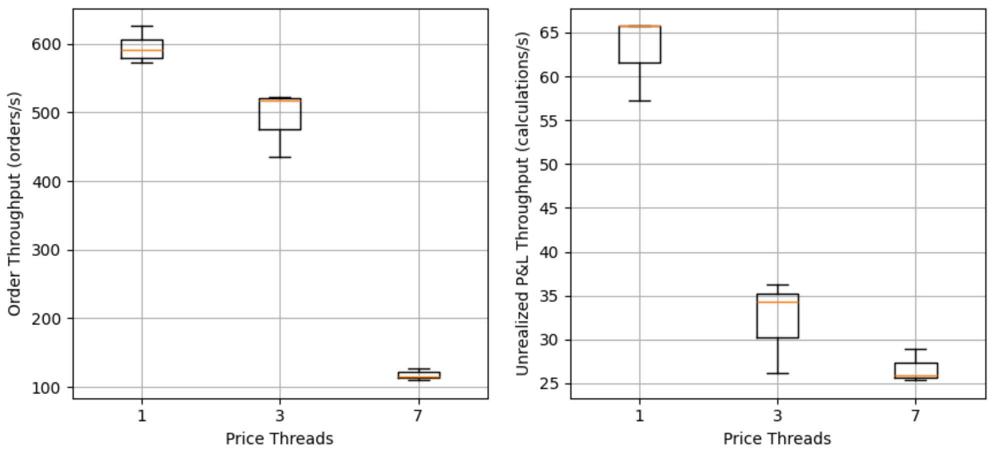

Figure 9: Investigating the effect of the number of price threads on Order & Unrealized P&L Throughput. Running on system node **S1** and database node **D1**.

Table 8: Average CPU usage of Monolith & Database, when running a Monolith with 7 order threads and 1, 3 and 7 price threads. Running on system node **S1** and database node **D1**.

Price Threads	Monolith CPU	Database CPU
1	12,67%	17.29%
3	33.37%	16.37%
7	98.12%	10.86%

Effect of Number of Order Threads | ***Performance Tuning of the Monolith***
As seen in Table 8, when running with a single price thread, the majority of CPU resources were not used. Based on this, the next logical step was to increase the order threads in order to utilize more of the CPU and investigate how this affected both throughputs.

For these test runs, there are four configurations, 1, 2, 3 and 4 threads per vCPU, that is, 7, 14, 28 and 56 order threads. Here, 1 per vCPU is the configuration that has been used during previous price thread test runs.

There is a clear performance gain for order throughput when two order threads are run per vCPU compared to one, but adding additional threads seems to only worsen it, as seen in Figure 10. Moreover, the P&L throughput is somewhat constant, probably because the order threads are mostly I/O dependent and do not interfere too much with P&L calculations.

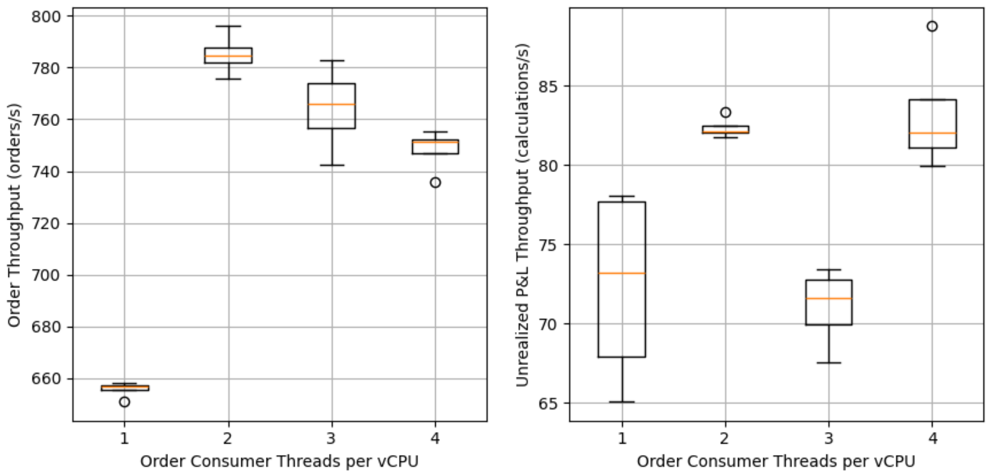

Figure 10: Investigating the effect of the number of order threads per vCPU on Order & Unrealized P&L Throughput. Running on system node **S1** and database node **D1**.

Result & Discussion | ***Performance Tuning of the Monolith***
To conclude the monolithic tuning, we can see that increasing the price threads only worsens the performance. This could be caused by added context switches when performing the P&L calculations, however since the database CPU also decreases with added price threads, as shown in Figure 8. It could be that the database is optimizing the P&L calculations better, when they are all coming over the same database connection.

Furthermore, adding additional order threads gives better order throughput, which would be expected since the order flow is database-heavy and should benefit from more database connections working in parallel and utilizing more of the database.

Finally, the configuration which seems most optimal is two order consumer threads per virtual CPU core and a one price consumer per seven vCPUs, that means one price thread being added at each “scale step”.

5.1.2 Performance Tuning of the Microservice

In this section we will try to tune the microservice to a near-optimal configuration. The initial focus is the number of pods/instances per service and the resource assignment between each service; this means that all pods of the same service will have the same amount of resources. The following steps will be performed and their results will be investigated in order to find the near optimal configuration;

- *Investigating the effects of gRPC*

- *Investigating effects when pushing resources towards consumers*
- *Investigating the use of Istio*
- *Investigating the use of threads in order consumers*
- *Result & Discussion of microserivce performance Tuning.*

Effects of gRPC communication | *Performance Tuning of the Microservice*
Since these services communicate using gRPC, they start a new background thread for each incoming gRPC request. Thus, they can handle as many parallel requests as they are allowed to create new background threads. Furthermore, since most of the services handle incoming gRPC requests by making SQL calls. The services act primarily as intermediaries, transporting data to and from the database. Meaning we can make the assumption that most of the services are quite I/O dependent.

This means that the gRPC background threads will have low CPU usage. Note that to utilize this, these services had to have access to multiple SQL connections, which is possible through the SQL connection pool mentioned in Section 4.5. Without it, they would instead only be able to make a single SQL call at a time.

Based on this reasoning, some initial microservice test runs were performed in which each service only had one pod/instance, and all pods had the same amount of resources. In Datadog we can that services such as Instrument Service, Account Service, et cetera. (Services that mostly act as DB entry points) can spend up to 85% of its time waiting for a response from the database and, as expected, do not fully utilize the CPU resources given to them.

The conclusion made from this result was that the biggest bottleneck to the order throughput was how many requests the order consumer could insert into the rest of the system. Based on this, additional tests need to be made with different pod instance configurations to see if increasing the order consumer's resources would improve the order throughput.

Effects of Prioritizing Consumers | *Performance Tuning of the Microservice*
When we are testing the effects of increasing the order consumer's resources, we will also increase the price consumer's resources. This is done because there are time constraints, meaning that we will not be able to test every configuration of price and order consumers. However, since the order and P&L throughput are equally important, keeping order and price consumers at the same instance count can be tolerated.

Four configurations will be used to test whether prioritizing resources to our consumers increases performance. Each configuration gradually increases the number of order and price consumers, while decreasing the rest of the services, as shown in Table 9. Furthermore, the price consumer pods have more CPU resources than the rest of the services, as shown in Table 10. This decision was made since previous test runs indicated that most of the demanding CPU work was performed when calculating option prices inside the price consumer/service.

Table 9: Instance count for each service, in four different configurations, when investigating increasing resources given to order and price consumers. Using database node **D1**.

Service	**Config 1**	**Config 2**	**Config 3**	**Config 4**
OrderConsumer	4	7	7	10
PriceConsumer(Service)	4	7	7	10
OrderService	4	3	4	2
PositionService	4	3	4	2
PnlService	4	3	4	2
AccountService	4	3	2	2
InstrumentService	4	3	2	2
PriceFetchService	4	3	2	2

Table 10: Resources for each service, in four different configurations, when investigating increasing resources given to order and price consumers. Using database node **D1**.

Configuration	**Service**	**CPU**	**Memory**
Configuration 1	PriceConsumer(Service)	180m	750MiB
	All other services	110m	750MiB
Configuration 2	PriceConsumer(Service)	150m	750MiB
	All other services	110m	750MiB
Configuration 3	PriceConsumer(Service)	150m	750MiB
	All other services	110m	750MiB
Configuration 4	PriceConsumer(Service)	138m	750MiB
	All other services	110m	750MiB

The result of these test runs shows that both order and P&L throughput increases when prioritizing resources for the order and price consumers, as shown in Figure 11. As expected, more order consumers created more total work in the system, and more price consumers also increased the resources that could be used for the P&L calculations. This meant that configuration four became the blueprint for further optimizations.

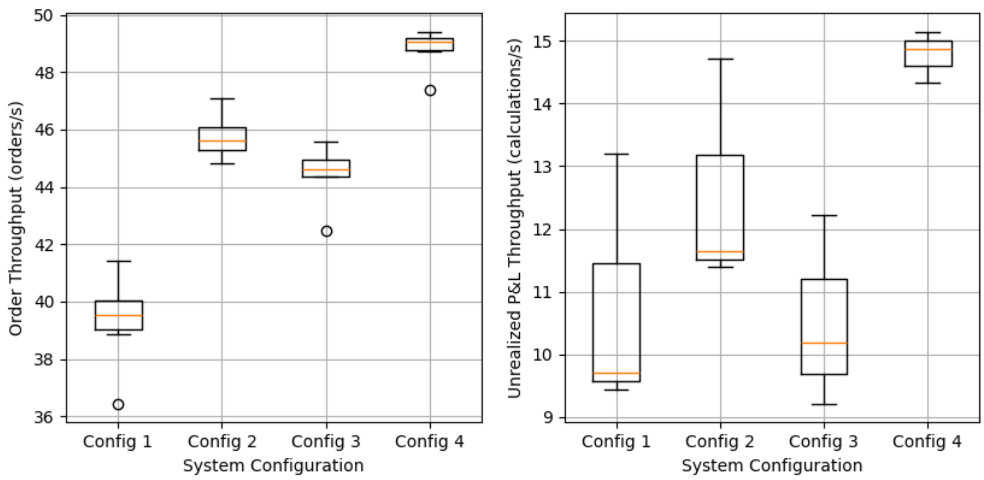

Figure 11: Investigating the difference between configurations from Tables 10 & 9 on Order & Unrealized P&L Throughput. Running on system node **S1** and database node **D1**.

Effects of Using Istio | *Performance Tuning of the Microservice*

The next factor that could limit the microservice architecture is the use of Istio. For each pod, the Istio service mesh uses *100m* CPU and *128MiB* Memory for running their sidecar container. This means that the configurations of Table 9 (all of which run 32 pods) allocate almost half of the CPU resources available to Istio. Based on this, fewer and larger pods could be more resource efficient than running a lot of smaller ones. To test this a cut-down version using fewer pods, but still biasing CPU towards consumers, will be tested against the "old" configuration from the previous tests. These configurations can be seen in Tables 11 and 12.

Table 11: Instance count for each service, in two different configurations, when investigating the usage of fewer pods. Using database node **D1**.

Service	Config 1	Config 2
OrderConsumer	10	4
PriceConsumer(Service)	10	4
OrderService	2	1
PositionService	2	1
PnlService	2	1
AccountService	2	1
InstrumentService	2	1
PriceFetchService	2	1

Table 12: Resources for each service, in two different configurations, when investigating the usage of fewer pods. Using database node **D1**.

Configuration	Service	CPU	Memory
Configuration 1	PriceConsumer(Service)	138m	750MiB
	All other services	110m	750MiB
Configuration 2	PriceConsumer(Service)	500m	1800MiB
	All other services	360m	1800MiB

The result can be seen in Figure 12, where a slight increase in throughput can be seen when running with fewer pods. Notably, even though we reduce the resources Istio uses by 50%, it is still taking about 25% of the total system resources.

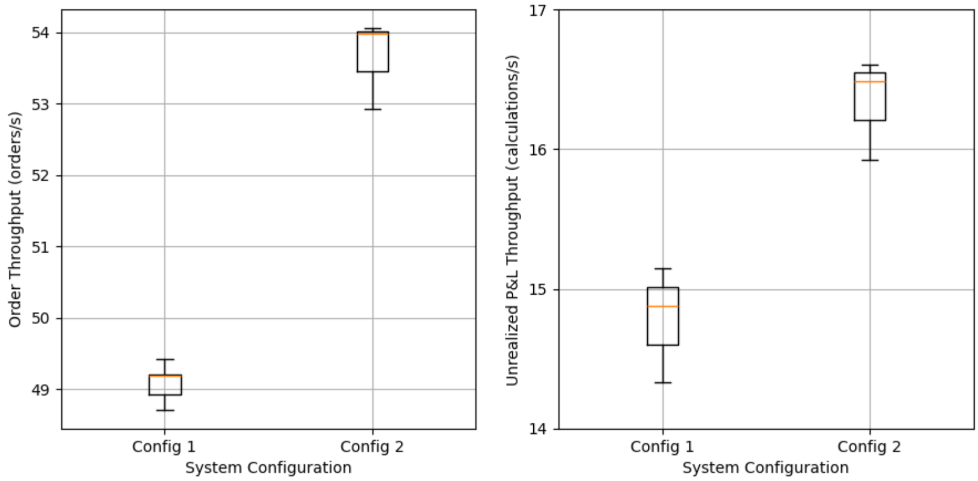

Figure 12: Investigating the difference between configurations from Tables 11 & 12 on Order & Unrealized P&L Throughput. Running on system node **S1** and database node **D1**.

From further investigation, it can be seen that the order consumer pods in configuration 2 only ran at around 20% CPU utilization, the price consumer at 70%, while the rest of the pod ran with between 40% and 30% CPU utilization. This behaviour could be tied back to the argument presented earlier: order consumers were not inserting enough work into the rest of the system, which meant a lot of idle services. Therefore, if we could create more work for the system, we could perhaps better utilize the system resources. We could accomplish this by running multiple threads in the order consumers and process multiple orders in parallel.

Effects of Threads in Order Consumers | *Performance Tuning of the Microservice* Comparing the result of configuration 2 that we can see in Figure 12, with the result of using threads, seen in Figure 13. A clear performance gain can be seen in order throughput, when adding threads, however, the performance gain starts to shrink when we run with 16 and 32 threads. The P&L throughput does not increase, which is expected, as threads in the order consumers should not positively affect the P&L. Instead, we can see a clear decrease when running with 16 and 32 threads, indicating some kind of congestion.

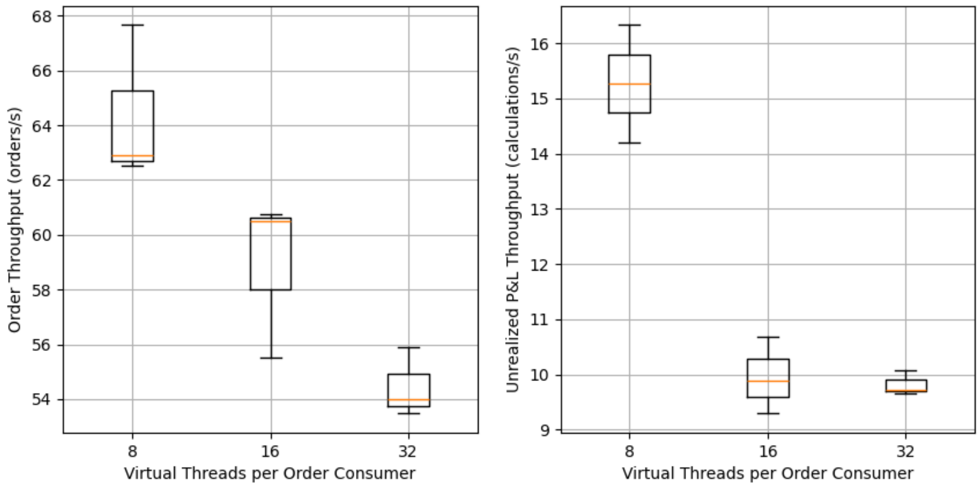

Figure 13: Investigating the effect of using threads in the order consumers on Order & Unrealized P&L Throughput. Running on system node **S1** and database node **D1**.

The performance gain here is quite small; the belief was that the increased order flow from running with threads should have made us utilize the resources at the other services better. This was not the case, and looking at the profile data in Datadog, the overall CPU utilization of the system was not close to 100%, that is, the added threads did not exhaust the system resource as we thought. Furthermore, the CPU usage of the database was only around 5%, indicating that the database was not the problem.

During test runs without running any price generator, meaning no unrealized P&L calculations, the order throughput increased to *130 Orders/s*, for 8 virtual threads, and overall CPU utilization increased. Indicating that the P&L calculations created some kind of congestion for the order flow, preventing us from fully exhausting the resources. Looking at gRPC call latencies, the time for fetching prices via the price fetch service doubled when running with P&L calculations. Moreover, the overall time for any gRPC call that went to the order and position service also increased.

The increased latency of the price fetch service could have resulted from the fact that the P&L calculations had to use the price consumers to calculate the option prices. When the order dataflow had to fetch prices, the price fetch service had to wait for the correct price consumer to process the gRPC call. This took time as the P&L calculations took most of the CPU cycles available at the price consumer. The increase in order and position gRPC latency is probably caused by the added SQL traffic when updating the unrealized P&L, which is expected.

Result & Discussion | ***Performance Tuning of the Microservice***
To conclude the tuning of microservices, prioritizing resources for consumers seems to achieve better performance than evenly distributing the resource across the system. Indicating that the resource distribution will be heavily tied to the specific system and its dataflow.

Furthermore, in addition to the added latency created by having multiple services communicating, Istio also takes a lot of resources. The main problem here is that the resources taken by Istio are directly connected to the number of pod instances, which becomes a problem when scaling by adding more pods. This is somewhat solved by running fewer but larger pods; however, if these pods become too big and too few, this could defeat some of the purposes of running microservices.

Another problem the microservice seems to have is congestion points being created, by both dataflows going through the same services. For example, both order and P&L uses the position service to update positions and the price fetch service to fetch prices.

Moreover, the low performance compared to that of the monolith could be connected to some extent to the use of Istio and the latency it adds when services are communicating. Although the microservice will always have quite a lot more total TCP latency than the monolith. The added TCP latency from Istio will be much greater for the microservice, as each TCP packet has to go through the Istio sidecars. That is, each gRPC call between services does the following:

$$ServiceA- > SidecarA- > SidecarB- > ServiceB$$

That is, three TCP jumps instead of one; this becomes clear when looking at the overall latency of the TCP packets. The monolith had 0.7 milliseconds TCP latency (to the database) and an additional 0.6 milliseconds from the proxy communication. At the same time, the microservice, when running with 8 virtual threads, had around 4 milliseconds of TCP latency and an additional 11 milliseconds from the proxies.

Finally, the most optimal configuration appears to be *Configuration 2* in Tables 11 and 12. Additionally, each order consumer uses 8 internal threads, to handle orders in parallel. Finally, all services that act as database entry points will utilize a database connection pool with a size of 32 connections.

5.1.3 Scaling Monolith

When scaling the monolith, it will be configured based on the discoveries made in Section 5.1.1, which is two order consumer threads per virtual CPU core and a one price consumer per seven vCPUs. See Sections 4.7 and 4.9 for the exact configuration of the monolith. Furthermore, the monolith will be scaled using the nodes described in Table 4. Finally, the monolith scaling will be performed in three steps, each step indicating an increase in the database. They are:

- *Scaling monolith with database node **D1***
- *Scaling monolith with database node **D2***
- *Scaling monolith with database node **D3***
- *Summary of monolithic scaling*

Scaling Monolith with Database Node D1 | *Scaling Monolith*
The first scaling measurement will use database node **D1** from Table 5. Here, an initial increase in both throughputs can be observed; however, both throughputs had decreased in the fourth step, as seen in Figure 14. When inspecting the CPU utilization provided by Datadog, the database CPU utilization was already at 90% at node 1, at nodes 2, 3, and 4,

it was at 100%. That is, the database prevents further performance improvements from only scaling the monolith. This means that we have to scale up the database in order to achieve additional performance improvements; therefore, the same measurement needs to be rerun with a bigger database node.

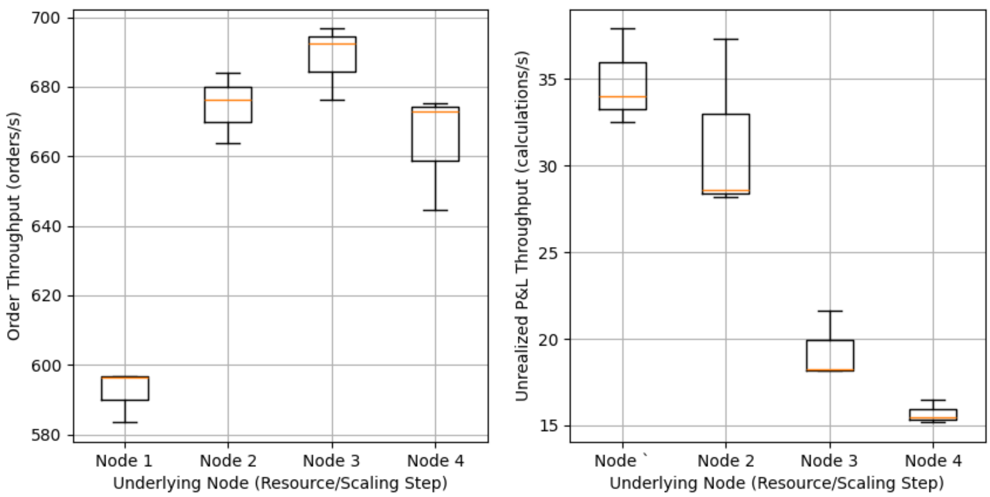

Figure 14: Investigating the effect of scaling the Monolith with database node **D1** on the Order & Unrealized P&L Throughputs.

Scaling Monolith with Database Node D2 | ***Scaling Monolith***

In the second scaling measurement, only the Nodes that was limited by the **D1** database is used. That is, Node 2, 3 and 4, will be rerun with database node **D2**, from Table 5. The result of this measurement can be seen in Figure 15. Here, a huge improvement in order throughput can be seen, compared to the previous measurement in Figure 14. However, the improvement begins to shrink at Node 3, because the database resources are being maxed out. Furthermore, while the P&L throughput is improving it is a lot slower than the order throughput. This means that we again need to scale up the database in order to achieve additional performance improvements.

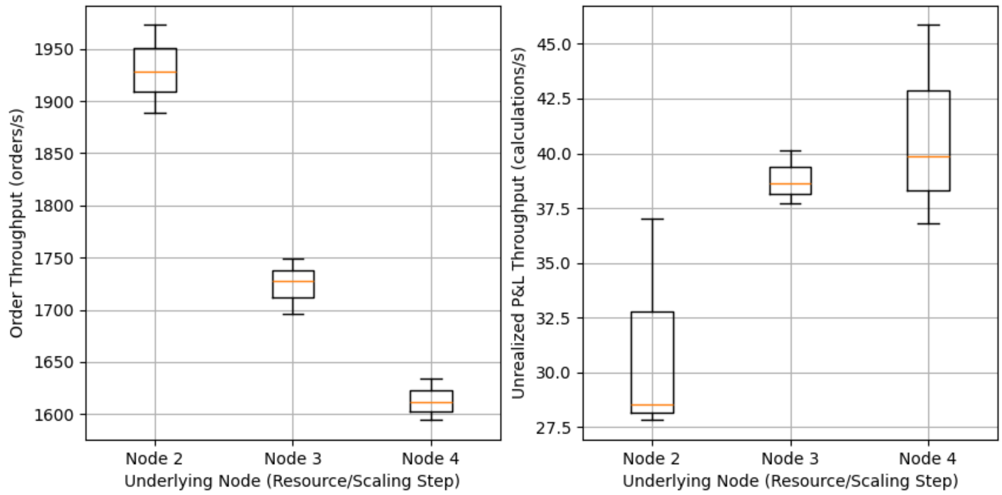

Figure 15: Investigating the effect of scaling the Monolith with database node **D2** on the Order & Unrealized P&L Throughputs.

Scaling Monolith with Database Node D3 | *Scaling Monolith*

The third and final scaling measurement will once again only use the nodes that was limited because of the database, that is Node 3 and Node 4 will be rerun with database node **D3**, from Table 5.

The result of the third and final measurement can be seen in Figure 16. The result shows that the order throughput increased from the previous measurement seen in Figure 15. However, the increase in order throughput between Node 3 and Node 4 was relatively small, as the database runs out of resources when we run on Node 4.

The P&L throughput does have an small increase, between Node 3 and Node 4, when looking at Figure 16. However, if we look at Figure 15, we can see that there is no increase in P&L throughput from running with a larger database node. This is expected as the P&L should be heavily CPU dependent and not bottleneck due to the database.

In order to get a better overview of the whole scaling and identify potential bottlenecks, the next step is to merge these measurements into a single graph.

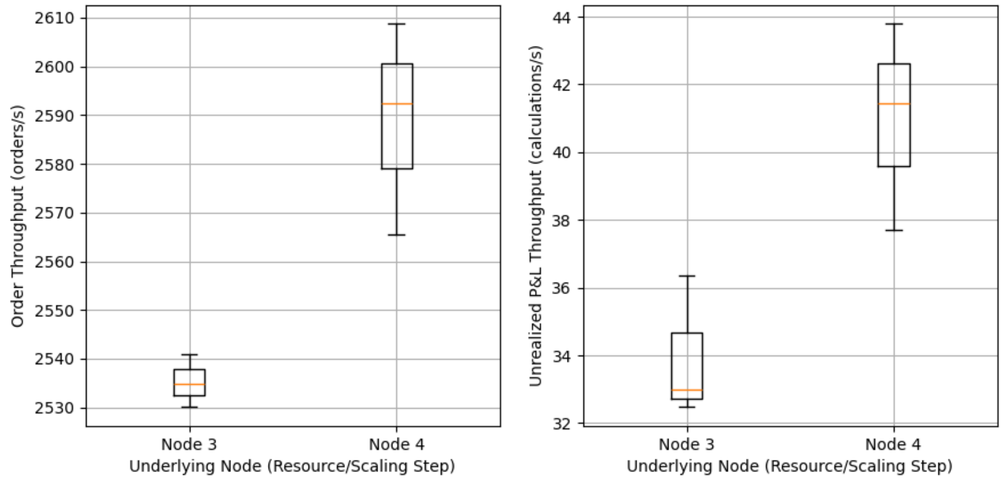

Figure 16: Investigating the effect of scaling the Monolith with database node **D3** on the Order & Unrealized P&L Throughputs.

Summary of Scaling | *Scaling Monolith*

To see an overview of the improvement in performance in order throughput, when scaling, we can look at Figure 17. Here, a clear increase in order throughput can be seen, which as pointed out earlier seems to be heavily connected to the database resources available.

The performance in P&L throughput when scaling, can be seen in Figure 18. Here, we can see that the P&L throughput seems to be less predictably than the order throughput. When running with the first database **D1**, the P&L decreases drastically when scaling the system node, but with the **D2** database, the P&L throughput instead increases. Moreover, there seems to be no real performance gain for the P&L when scaling up database resources. However, this is somewhat expected as the P&L is heavily CPU dependent.

To further understand the monolith's and our P&L calculation's behaviour, we have to look into the profiler data generated during the measurements. Starting by looking into the monolith running on Node 3 with database **D2** versus database **D3**. Where an increase in order throughput could be observed, but not in the P&L calculations. Firstly, when running with database **D2**, the wall-clock time spent on the P&L calculations was lower than when running with database **D3**. That means that the calculations became slower as more database resources were added. Furthermore, as expected, the monolith running with database **D3** had lower I/O time (Blocking I/O time) than the **D2** database. This seemed to show that the added I/O time allowed the P&L calculations to utilize more of the CPU cycles, but when the I/O time was reduced, the order consumer reclaimed a piece of these CPU cycles. That is, less I/O blocking meant higher CPU usage by the order threads, which could have led to more contention and context switching(interrupting P&L calculations), which in turn resulted in decreased P&L performance.

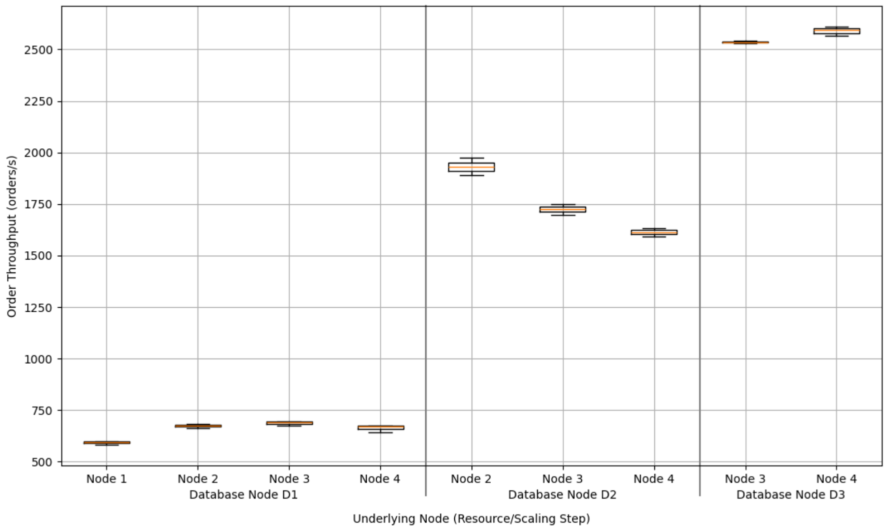

Figure 17: Order Throughput when scaling the monolith. Includes measurements for each database node.

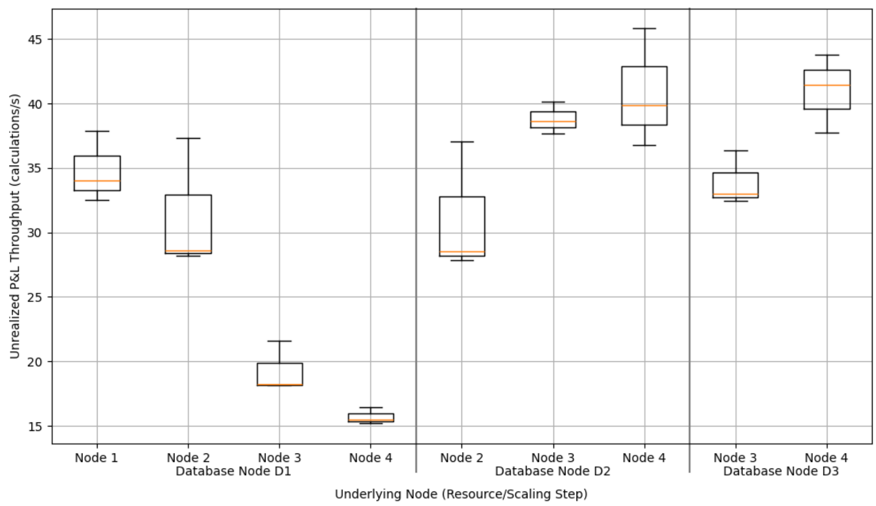

Figure 18: Unrealized P&L Throughput when scaling the monolith. Includes measurements for each database node.

However, the behaviour seen when scaling the system using database **D1** contradicts this argument, that is the P&L is decreasing instead of increasing as I/O time increases. Looking further into the data available at Datadog, the TCP latency to the database provided an explanation. Table 13, shows the TCP latency of the database and the CPU utilization of the database. From which it can be seen that when the monolith was scaled up using database **D2**, the TCP latency increased more slowly than when scaling on database **D3**. This indicates that database utilization was high enough on node **D1** that the P&L calculations started to bottleneck due to SQL call latency rather than due to the available CPU cycles.

Table 13: Database TCP Latency and CPU utilization for databases **D1** and **D2**, when scaling the monolith.

System Node	Database Node	TCP Latency	Database CPU utilization
S1	D1	1.31ms	100%
S2	D1	3.05ms	100%
S3	D1	5.03ms	100%
S4	D1	7.14ms	100%
S2	D2	1.19ms	87%
S3	D2	2.11ms	100%
S4	D2	2.82ms	100%

5.1.4 Scaling Microservice

When scaling the microservice, it will be configured based on the discoveries made in Section 5.1.2. That means that it will run with *Configuration 2* in Tables 11 and 12. Where each order consumer uses 8 internal threads, to handle orders in parallel. Services that act as database entry points will utilize a database connection pool with a size of 32 connections. See Sections 4.7 and 4.9 for the exact configuration of the microservice. The microservice will be scaled using the nodes described in Table 4. Finally, the microservice scaling will be performed in three steps, each step indicating an increase in the database. They are:

- *Scaling microservice with database node **D1***
- *Scaling microservice with database node **D2***
- *Scaling microservice with database node **D3***
- *Summary of microservice scaling*

Scaling Microservice with Database Node D1 | ***Scaling Microservice***

The first scaling measurement will use database node **D1** from Table 5. Here, a clear performance gain can be seen for both order and P&L throughput as more resources are added, as seen in Figure 14. However, while order throughput's rate of improvement seems to slow down, the P&L improvement accelerates. This could be explained by the fact that the P&L flow is more CPU-bound, while the order flow is more I/O-bound to the database.

When looking at the CPU usage from Datadog, at each node. It can be seen that at Node 2, the CPU utilization of the database is just below 100%, and at Node 3, the CPU utilization was maxed out. An interesting observation as Node 3 still had almost double the throughput from Node 2, indicating that the database performance does not affect the microservice nearly as much as the monolith. Still, the next step will be to scale up the database in order to further verify this.

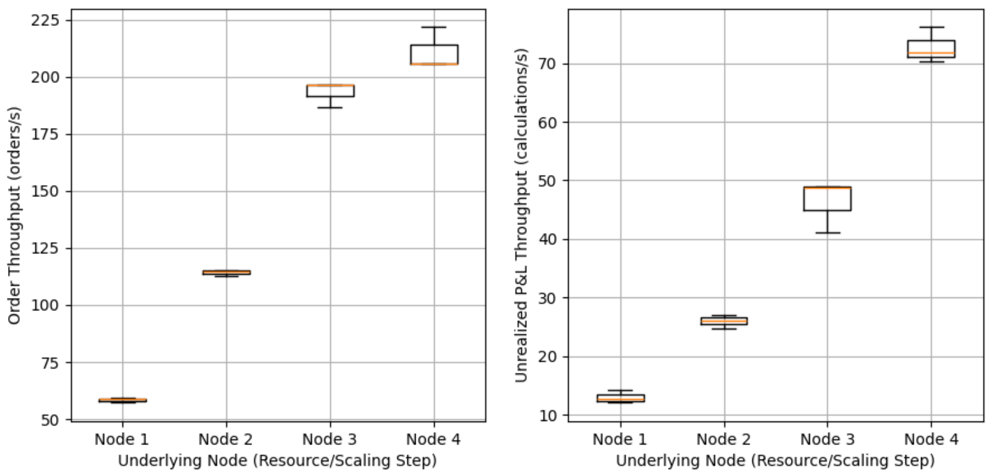

Figure 19: Investigating the effect of scaling the Microservice with database node **D1** on the Order & Unrealized P&L Throughputs.

Scaling Microservice with Database Node D2 | ***Scaling Microservice***
As in the monolith the second microservice measurement will only use the nodes that was limited by the database, that is Node 3 and Node 4 will be rerun with database **D2**, from Table 5.

The result of this measurement can be seen in Figure 12. We can see that Node 3 increases both throughputs but not much, which further indicates that the database is not the limiting factor of the system. Looking at Node 4 we can see a performance increase of around 50%, however, Node 4 is also fully utilizing the database CPU. That means running Node 4 with the biggest database node could give even better performance.

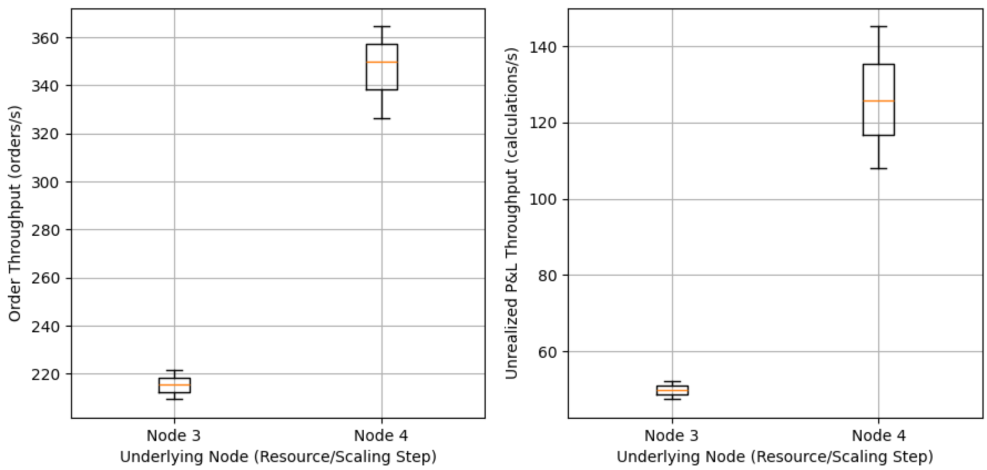

Figure 20: Investigating the effect of scaling the Microservice with database node **D2** on the Order & Unrealized P&L Throughputs.

Scaling Microservice with Database Node D3 | ***Scaling Microservice***

In the final microservice measurement only Node 4, will be rerun with database **D3**, from Table 5. The result can be seen in Figure 21. In which order throughput increased compared to Node 4 in Figure 21, when running with a bigger database node. However, the increase is quite small and, looking at the P&L, no increase can be seen at all.

To get a good overview and comparison of all measurements, they will next be merged into a single graph, the same as the monolith.

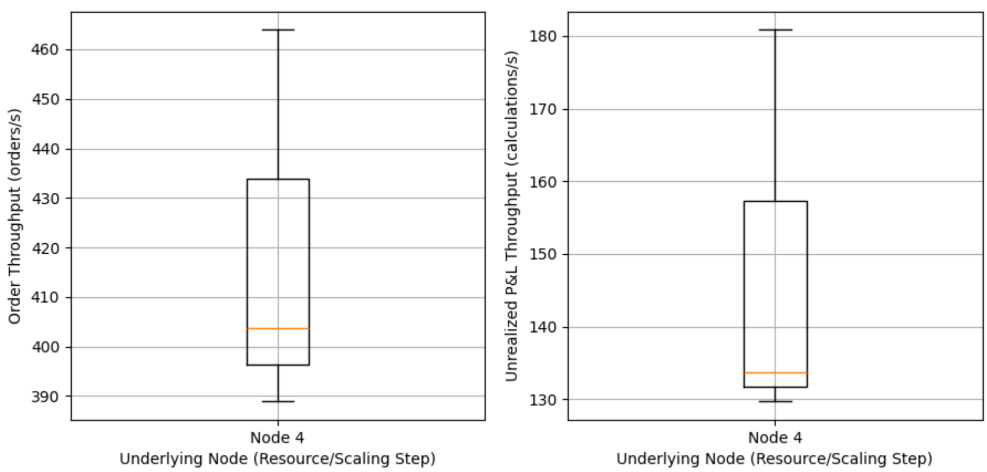

Figure 21: Investigating the effect of scaling the Microservice with database node **D3** on the Order & Unrealized P&L Throughputs.

Summary of Scaling | *Scaling Microservice*
An overview of the microservice scaling capabilities can be seen in Figures 22 and 23. It shows a clear performance increase as resources are scaled up, although the microservice seems to achieve most of its performance gain from added system resources and not database resources. For example, running the system on Node 3 has the same performance when using databases **D1** and **D2**, even though the **D1** database CPU was maxed out with Node 3. This behaviour was reasonable since the microservice used simpler SQL queries and performed most of the order processing within the system, compared to the monolith that utilized more complex SQL queries.

In summary, the microservice seems to be mainly limited by the amount of resources given to it and not by the database, which is somewhat worrying since Istio initially takes a lot of its resources. However, as long as more resources are provided and the database is not overloaded, the microservice seems to scale both throughputs evenly. This is important as the monolith struggled to scale the P&L, with the order flow taking all the resources.

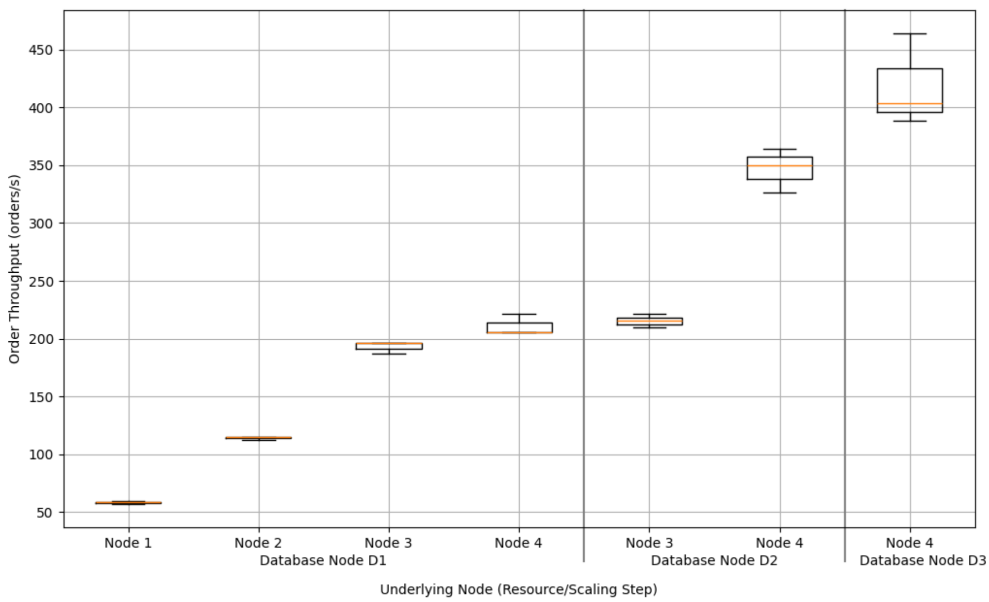

Figure 22: Order Throughput when scaling the microservice. Includes measurements for each database node.

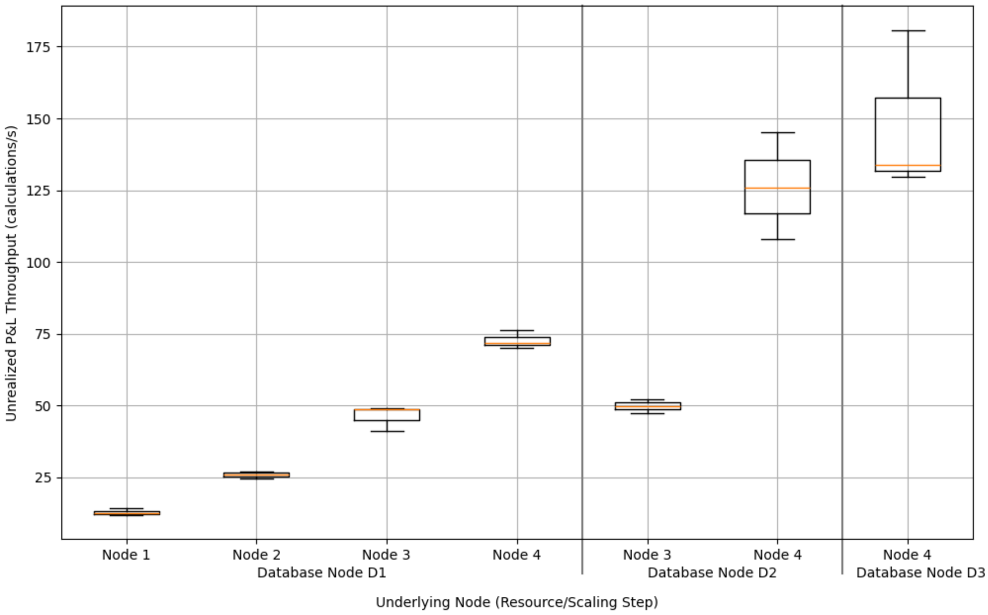

Figure 23: Unrealized P&L Throughput when scaling the microservice. Includes measurements for each database node.

5.2 Comparing Scalability(RQ1)

The following section presents the performance-to-cost functions using graphs. Starting by looking at the order throughput, which can be seen in Figure 24. The result shows that the monolith not only has a much higher starting throughput but also appears to initially scale better. This would be in line with what was expected, where the results of both [3] and [4] of Related Works (Section 3) showed that the monolith was better at the beginning.

However, the monolithic scaling seems to stagnate after some time. Instead, by looking at the percentage increase in throughput from the “starting” throughput, that is, the throughput at the first node/scaling step, which can be seen in figure 25. The microservice sees a better percentage improvement from its starting point compared to the monolith over time; furthermore, the monolithic scaling completely stagnates. Indicating that at some point the microservice performance will overtake the monolith, if enough resources are given. This is in line with the conclusion made by [3] and [4], where the microservice scales better than the monolith.

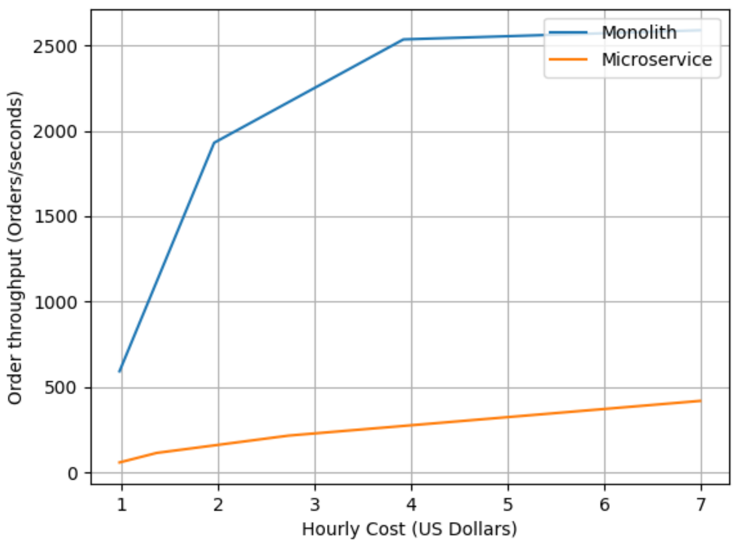

Figure 24: The Order Throughput of the monolith and microservice, mapped to the hourly cost of running the systems.

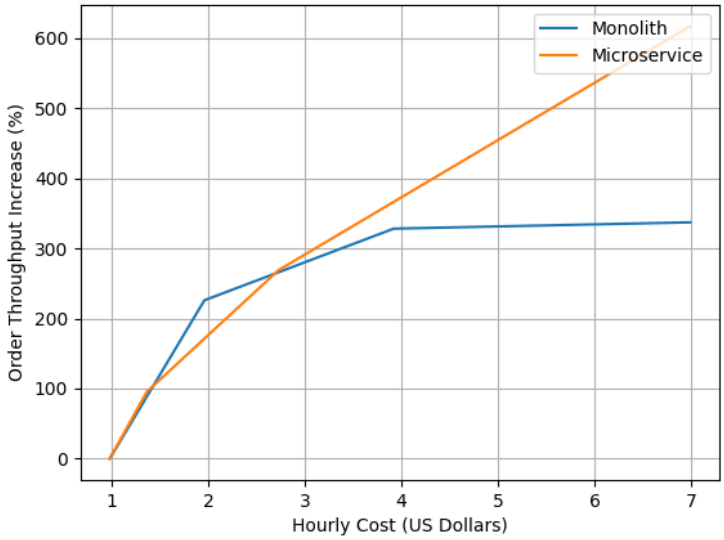

Figure 25: The percentage increase of the Order Throughput of the monolith and microservice, from initial scaling step (Node 1), mapped to the hourly cost of running the systems.

If we look at the P&L performance to cost function seen in Figure 26, the microservice is now scaling, while the monolith now achieves almost no performance increase. Looking at the percentage increase for the P&L (same as the order throughput), in Figure 27. We see the same behaviour as in ordinary P&L performance to cost function, the P&L throughput increased about 1000% when running with the microservice and virtual zero when running the monolith.

An interesting takeaway from these figures is that the microservice order flow is very low compared to the monolith, between 5 to 10 times worse. However, the microservice P&L flow is better, not initially, but scales, whereas the monolith does not. Showing the problems that can appear when running multiple dataflows in the monolith.

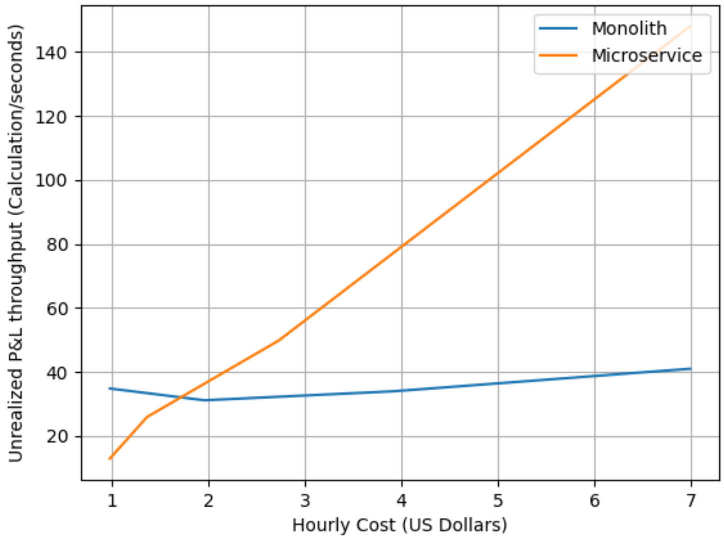

Figure 26: The Unrealized P&L Throughput of the monolith and microservice mapped to the hourly cost of running the systems.

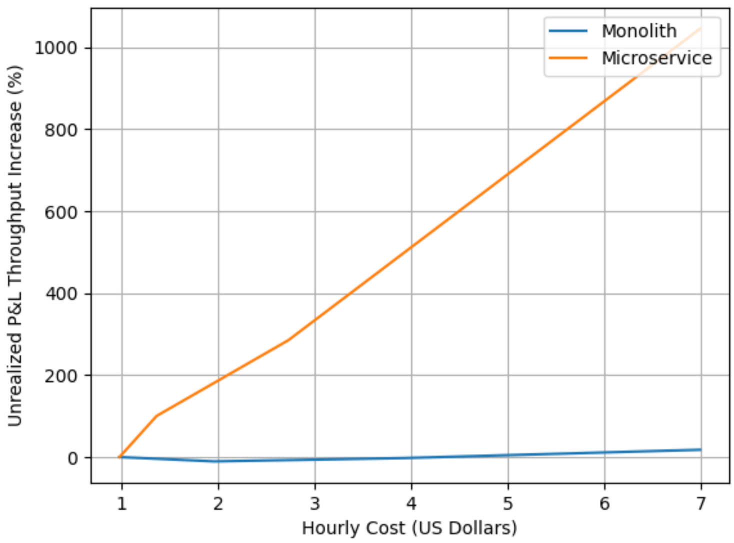

Figure 27: The percentage increase of the Unrealized P&L Throughput of the monolith and microservice, from initial scaling step (Node 1), mapped to the hourly cost of running the systems.

6 Discussion

In order to answer the research questions presented in Section 1.1. This section will analyze and discuss the result presented in Section 5. In Section 6.1, we will try to find potential bottlenecks in the systems that limit their scalability. Then, in Section 6.2, we propose guidelines that can be helpful in choosing a monolithic or microservice architecture.

6.1 Discovering Bottlenecks(RQ0)

This section will discuss the result of Section 5, trying to find potential bottlenecks that would limit the monolithic or microservice system's potential to scale up effectively.

6.1.1 Monolithic Architecture

Starting by looking at the initial investigation found in Section 5.1.1. The initial bottleneck limiting the monolith's performance seems to be the number of threads running compared to the available resources, and how these should be divided over both the order and P&L dataflow. Adding threads to the P&L data flow decreases the order throughput, as they are quite CPU heavy. However, it also decreases the P&L throughput, indicating that added context switches are worsening the performance. Furthermore, adding more threads to the order flow increases the order throughput and has almost no effect on the P&L. That is, the data flows are affecting each other in different ways.

This in turn prevents us from effectively increasing both throughputs when scaling the system and effectively dividing the available resources over multiple dataflows. Meaning, a monolith seems to favour a simpler design with possible at most a single dataflow. As adding more dataflows and making systems more complex, will lower our possibility to control and distribute resources. This means that a monolith architecture could be ideal for a system that only runs the order flow.

Moving on to the scaling result and the limiting factors that have been observed in Section 5.1.3, we can see some interesting factors. Firstly, the monolithic architecture seems to be a valid approach for simpler systems for quite a while. Although there is a lot of data running through the system, it is scaling well for a single data flow. Secondly, it is clear that the majority of order throughput increases come from increasing the database resources. This could be expected, as the monolith utilizes more complex queries and performs many of its calculations within the database. However, at some point, problems will arise when scaling the database, and only increasing the underlying node size as we have done, will not be feasible. There will be too many concurrent connections and active tables. At that point, we could start to scale the database by, for example, sharding it into multiple instances (each instance having a unique area of responsibility). However, this would potentially drive up the cost and worsen the performance-to-cost function.

Looking past the database scaling limitations, there is still almost no increase in P&L throughput, even though we are going from 7 vCPUs to 64 vCPUs. This is a bit unexpected since the P&L throughput is quite CPU heavy, and one would think that it would benefit from the increased vCPUs available. This further proves the point made earlier, that is, it is difficult to effectively divide the available resources over multiple dataflows. Although more vCPUs are available, it is almost impossible to ensure that the P&L calculations are getting these, as it is the operating system that decides which threads are to run. Here, it seems that the constant context switching coming from the constant IO calls made by the order threads is negatively affecting the price thread's chances to effectively utilize the available resources.

6.1.2 Microservice Architecture

Looking into potential bottlenecks and limits in the microservice, there does not appear to be a significant bottleneck that restricts the microservice from scaling up. As seen in Section 5.1.4, the microservice mainly scales when the system receives more resources. Adding resources to the database increases both throughputs, but is nowhere close to the monolithic architecture. This would be expected since the microservice utilized the database less than the monolith.

However, while the microservice sees a good performance increase compared to its start value, the performance increase is much lower than that seen in the monolith, as the initial "start" throughput is much lower. In Section 5.1.2, this is investigated. One reason for this seems to be the use of Istio, which has been injecting an additional sidecar container into each pod. This is done to enable mTLS between services, increase security, and resemble a "real" system used in the industry. However, this is not only taking a lot of resources from the microservice system. It also adds a large amount of extra TCP latency between the services, multiple times larger than for the monolith.

Finally, there is also the aspect of how much resources each service instance/pod should be given to utilize the available resources the best. This is one of the aspects that have not been investigated enough in this thesis; however, some conclusions can still be drawn. Firstly, it is very difficult to know how to best scale a microservice system. Secondly, the amount of resources that each service instance should have is extremely system-specific and needs to be determined for each system. However, based on the results when running the microservice, it is clear that the data flows are affecting each other heavily as they are working on the same data. This creates congestion points where some services are placed under a much greater load than the rest of the system. Lastly, it seems to be favourable to push resources towards the services which consumes/fetches the work, which the rest of the system will handle. However, this is only applicable if the other services utilize a thread pool to handle multiple requests at once and are mostly I/O-bound.

Based on this, a good approach when configuring resources of the services could be to initially push resources towards these work-consuming services, in order to produce enough work to exhaust the rest of the services. Then it would be easier to identify the services which are becoming congestion points/bottlenecks, and start to modify the resources for the services based on their workload.

6.2 Comparing Scalability(RQ1)

When comparing the scalability we are not interested in a single throughput and its performance-to-cost function, but rather in how the architectures perform over both the order and P&L dataflow. Based on the result presented in 5.2, the monolith does not appear to be able to scale both data flows simultaneously in any cost-effective manner. Although its order throughput has a good performance-to-cost function (initially), its P&L performance-to-cost seems to be almost static. Furthermore, the monolith's order throughput scaling slows down quite quickly at 2 dollars and almost completely stagnates at around 4 dollars, as seen in 24. This behaviour is linked to the steps at which we scale the database, indicating the monolith's reliance on its database and also showing the monolith's problems when scaling simply by increasing system resources.

On the other hand, the microservice has a completely different behaviour, as the microservice is capable of scaling both dataflows in a relatively cost-effective manner. Furthermore, from the results we have, there does not seem to be a point in the near future where the scaling stagnates. However, the problem with the microservice is that the initial order throughput is so low that it cannot catch up with the monolith. This behaviour is more easily observed in Figure 25, where the microservice achieves a better percentage increase (from its initial performance) compared to its cost than the monolith, when it reaches a cost of approximately 3 dollars per hour. Indicating that, if not for its low initial performance (performance on system node 1), the microservice might have had a better performance-to-cost function for the order throughput than the monolith.

Based on this, it seems clear that there does not exist a universal answer when one architecture is more cost-efficient than the other. Nevertheless, this discussion can tell us in which systems, dataflows, and environment one architecture would be preferred over the other.

The monolith appears to favour a simple system design, likely with only a single dataflow, as managing multiple dataflows can lead to resource allocation issues. Furthermore, running dataflows that utilizes the heavily optimized database engine with multi-table queries and transactions, will be better with the monolith. Although the monolith's scaling seems to stagnate at some point (looking at the order throughput), the initial performance advantage might be enough to run the monolith over the microservice, even when scaling the system heavily. Lastly, since a lot of the monolith performance gains come from the scaling of its database, it will be crucial to deploy and run the database in a way that allows it to scale up.

Moving on to the microservice, it seems to be better suited for a multi-dataflow system. As it allows the developer to have more control over how much resources each dataflow will have access to, it also simplifies the process of benchmarking and optimizing individual dataflows. However, this can then worsen one of the dataflows in favour of making their performance equal, which was what happened with the order throughput at the microservice. This means that if one throughput has important performance criteria to meet, running multiple dataflows in a microservice might not be the best choice. Furthermore, running multi-table queries and transactions from the microservice, will often not be as efficient as in the monolith. As a single service, might not have access to all the needed tables and therefore have to communicate with other services and aggregate multiple SQL calls locally.

An important point to take into consideration is the size of the potential microservice, that is, how many instances are there for each service and what their resources are? If the pods are quite small, Istio will take a lot of resources from the system. Then it might be better(in terms of cost) to investigate running bigger pods or simply running that microservice as a monolith. We could run some kind of combined architecture, in which workflows that utilize database transactions and read-heavy multi-table lookups/writes can be run in a monolith, while workflows and services that are, for example, compute bound could be spin out into microservics. This would further increase our ability to control and distribute resources correctly, while we could limit the microservice to only use large pods, minimizing the resources taken by Istio.

The result of this thesis somewhat contradicts the conclusions made by [3] and [4] of Related Works (Section 3). They state that even though the monolith initially is better, the microservice takes over after a certain point. This difference in behavior could be due to the fact that our microservice system is larger and has more services, increasing the communication needed, which in turn pushes the point at which the microservice overtakes the monolith further away.

7 Conclusion & Future Work

In this thesis, we have looked at a vertically scaling monolithic architecture and a mostly horizontally scaled microservice architecture and how they behaved when scaling up. The work has focused on workflows that are commonly found in financial systems, characterized by high throughput and low latency. Based on this, a system was developed using a monolithic and microservice design/architecture. It handles incoming buy and sell orders and consumed external price data using two different dataflows, one more I/O-intensive and one more CPU-intensive. The work initially looked at what limits the systems from effectively scaling up, trying to find bottlenecks that limited the architectures. It then moved on to try to answer when one architecture should be chosen over another based on their performance to their cost of running.

The result of this showed that the monolithic architecture was heavily dependent on the database, having great performance gains when scaling up the database resources. However, as the monolith scaled, the I/O-intensive dataflow overtook the CPU-intensive, resulting in quite good I/O-intensive dataflow performance gains but with very poor CPU-intensive dataflow performance gains. Highlighting the difficulty in distributing resources over multiple dataflows when using a monolithic architecture. At the same time, the microservice became limited by the added IO latency partially introduced by the use of service mesh and the accompanying mTLS, but was also heavily dependent on how the resources were divided over each service.

Moreover, the results showed that the monolith had trouble scaling both dataflows effectively, the IO-intensive dataflow "took over" and scaled exceptionally well (compared to the microservice), while the CPU-intensive dataflow was almost static. On the other hand, the microservice was capable of scaling both dataflows and a point where scaling stagnation started could not be seen in the near future. However, the microservice's poor initial IO performance meant that it had trouble catching up with the monolith and came no where near the monolithic order performance. No universal answer could be found when one architecture is more cost-efficient than the other, but some conclusions could be drawn. The monolith seems to favour a simple system design with possibly at most a single dataflow, as problems arise surrounding resource allocation over multiple dataflows. The microservice seems to be better suited for systems with multiple dataflows, as it is easier to control resource allocation over multiple dataflows. However, problems can arise if one of the dataflows has important performance criteria to meet.

Finally, at some point, the monolith will stop scaling effectively, at which point some kind of microservice could become a better option. When this could be, would be different between different systems and is almost impossible to answer without testing with the affected system. However, since the monolith has clear performance to cost advantage for quite a while, it seems irrational to run a system as a microservice from the start simply based on this premise only.

7.1 Future Work

The following section will discuss potential extensions and future work on the result and conclusions made in this thesis.

7.1.1 Caches

A “real” production system would utilize some kind of cache such as Redis[15], to minimize the time to retrieve data from databases, etc. This could decrease the retrieval time from the database, since, for example, commonly used entries could be cached. It would be interesting to see how each architecture behaves when this is used and if one of them obtains a better performance to cost function from it.

7.1.2 Monolith & Sharding

It would be interesting to investigate how the monolithic performance to cost function is affected by allowing it to use “sharding”. That is, split the system into multiple instances, over, for example, Kafka partitions. This would allow a monolith instance that handles more popular instruments, which have more orders on them. To be given more resources, allowing for more effective resource-to-performance handling.

7.1.3 Increase Scale

This investigation tried to increase the scale on which the systems operated, using “real” test data, in the form of the internal test data used by Clear Street, and increasing the complexity of the system from previous papers presented in the *Related Work* Section 3. However, since there was a limited amount of time to accomplish this, the system might not be as optimized as it should be. That is, the systems are not performing at the scale that a “real” system would be, and it could be interesting to see how the systems start to behave if the order latency is decrease to actually mimic a real production system, with the same performance criteria.

7.1.4 Partitioning Database

For this thesis the PostgreSQL database has been a single instance which have been scaled vertically if needed, by running it on larger EC2 nodes. However, if the database was partitioned into multiple instances, we could possibly get better utilization of database resources, by only scaling specific instances. It would be interesting to see how the performance to cost functions are affected by different types of database partitioning and if anyone could improve our systems.

7.1.5 Hybrid Architectures

Another interesting perspective to look at would be the usage of “hybrid” architectures, that is, a system that incorporates key characteristics from both a monolithic architecture and a microservice architecture. For example, some components of the system might benefit more from vertical scaling, while other benefits more from horizontal scaling. Identifying these components, and scaling system components differently, might be more resource efficient.

Moreover, as mentioned, if some dataflows utilize database transactions and read-heavy multi-table lookups/writes, it might be favourable to run it as a monolith. While splitting other more CPU-intensive dataflows to microservices, to avoid the dataflows to interfere with each other and worsen each others performance.

[15]https://redis.io/

References

[1] *Amazon EC2 - Cloud Compute Capacity - AWS.* Amazon Web Services, Inc. URL: https://aws.amazon.com/ec2/ (visited on 02/27/2025).

[2] Sururah A. Bello et al. "Cloud computing in construction industry: Use cases, benefits and challenges". In: *Automation in Construction* 122 (Feb. 1, 2021), p. 103441. ISSN: 0926-5805. DOI: 10.1016/j.autcon.2020.103441. URL: https://www.sciencedirect.com/science/article/pii/S0926580520310219 (visited on 02/04/2025).

[3] N Bjorndal et al. *(PDF) Migration from Monolith to Microservices : Benchmarking a Case Study*. ResearchGate. Mar. 2020. DOI: 10.13140/RG.2.2.27715.14883. URL: https://www.researchgate.net/publication/339749917_Migration_from_Monolith_to_Microservices_Benchmarking_a_Case_Study (visited on 02/14/2025).

[4] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation". In: *IEEE Access* 10 (2022), pp. 20357–20374. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3152803. URL: https://ieeexplore.ieee.org/abstract/document/9717259 (visited on 02/14/2025).

[5] André B. Bondi. *Characteristics of scalability and their impact on performance*. DOI: 10.1145/350391.350432. URL: https://dl.acm.org/doi/epdf/10.1145/350391.350432 (visited on 02/04/2025).

[6] Greg Boss et al. *Cloud Computing IBM*. Oct. 8, 2007. (Visited on 02/07/2025).

[7] *Cloud Computing Market Size, Share, Growth & Forecast [2032]*. URL: https://www.fortunebusinessinsights.com/cloud-computing-market-102697 (visited on 02/04/2025).

[8] "Distributed Systems Definition". In: Maarten van Steen and Andrew S. Tanenbaum. *Distributed systems*. Third edition, Version 3.01. Erscheinungsort nicht ermittelbar: Maarten van Steen, 2017, p. 2. ISBN: 978-1-5430-5738-6.

[9] Nicola Dragoni et al. "Microservices: How To Make Your Application Scale". In: *Perspectives of System Informatics* (Jan. 2018), pp. 95–104. DOI: 10.1007/978-3-319-74313-4_8. URL: https://www.researchgate.net/publication/322543131_Microservices_How_To_Make_Your_Application_Scale (visited on 02/04/2025).

[10] EDA Summit. *Principles and Guidelines of Electronic Trading Systems Architecture*. June 13, 2023. URL: https://www.youtube.com/watch?v=-L2XIl0eXvY (visited on 02/27/2025).

[11] *Event-Driven Architecture*. Amazon Web Services, Inc. URL: https://aws.amazon.com/event-driven-architecture/ (visited on 02/27/2025).

[12] *Evolution of Financial Exchange Architectures*. InfoQ. URL: https://www.infoq.com/presentations/financial-exchange-architecture/ (visited on 02/24/2025).

[13] *F.30. pg_stat_statements — track statistics of SQL planning and execution*. PostgreSQL Documentation. Feb. 20, 2025. URL: https://www.postgresql.org/docs/17/pgstatstatements.html (visited on 02/27/2025).

[14] Martin Fowler. *bliki: Microservice Premium*. martinfowler.com. URL: https://martinfowler.com/bliki/MicroservicePremium.html (visited on 02/04/2025).

[15] Martin Fowler and James Lewis. *Microservices*. martinfowler.com. URL: https://martinfowler.com/articles/microservices.html (visited on 02/07/2025).

[16] Dennis Gannon, Roger Barga, and Neel Sundaresan. "Cloud-Native Applications". In: *IEEE Cloud Computing* 4.5 (Sept. 2017). Conference Name: IEEE Cloud Computing, pp. 16–21. ISSN: 2325-6095. DOI: 10.1109/MCC.2017.4250939. URL: https://ieeexplore.ieee.org/abstract/document/8125550 (visited on 02/07/2025).

[17] *Glossary of finance and debt: English-French-Spanish = Glossaire des finances et de la dette: français-anglais-espagnol = Glosario de finanzas y de deuda: español-inglés-francés*. A World Bank glossary = Glossaire de la Banque mondiale. Washington, D.C., U.S.A: World Bank, 1991. 213 pp. ISBN: 978-0-8213-1644-3.

[18] PostgreSQL Global Development Group. *PostgreSQL*. PostgreSQL. Feb. 28, 2025. URL: https://www.postgresql.org/ (visited on 02/28/2025).

[19] *gRPC Load Balancing on Kubernetes without Tears*. Kubernetes. Section: blog. Nov. 7, 2018. URL: https://kubernetes.io/blog/2018/11/07/grpc-load-balancing-on-kubernetes-without-tears/ (visited on 02/27/2025).

[20] *Introduction to Apache Kafka Partitions*. Confluent. URL: https://developer.confluent.io/courses/apache-kafka/partitions/ (visited on 02/24/2025).

[21] Miika Kalske, Niko Makitalo, and Tommi Mikkonen. "(PDF) Challenges When Moving from Monolith to Microservice Architecture". In: *ResearchGate* (Nov. 21, 2024). DOI: 10.1007/978-3-319-74433-9_3. URL: https://www.researchgate.net/publication/323312732_Challenges_When_Moving_from_Monolith_to_Microservice_Architecture (visited on 02/24/2025).

[22] Lin Sun (Solo io) Kohavi (Solo.io) Yuval. *Secure Application Communications with Mutual TLS and Istio*. Istio. URL: https://istio.io/latest/blog/2023/secure-apps-with-istio/ (visited on 03/14/2025).

[23] Peter Mell and Timothy Grance. "The NIST Definition of Cloud Computing". In: ().

[24] Miloš Milić and Dragana Makajić-Nikolić. "Development of a Quality-Based Model for Software Architecture Optimization: A Case Study of Monolith and Microservice Architectures". In: *Symmetry* 14.9 (Sept. 2022). Number: 9 Publisher: Multidisciplinary Digital Publishing Institute, p. 1824. ISSN: 2073-8994. DOI: 10.3390/sym14091824. URL: https://www.mdpi.com/2073-8994/14/9/1824 (visited on 02/07/2025).

[25] *O'Reilly's Microservices Adoption in 2020 Report Finds that 92% of Organizations are Experiencing Success with Microservices*. URL: https://www.oreilly.com/pub/pr/3307 (visited on 02/04/2025).

[26] *Overview*. Kubernetes. URL: https://kubernetes.io/docs/concepts/overview/ (visited on 03/03/2025).

[27] Ling Qian et al. "Cloud Computing: An Overview". In: *Cloud Computing*. Ed. by Martin Gilje Jaatun, Gansen Zhao, and Chunming Rong. Berlin, Heidelberg: Springer, 2009, pp. 626–631. ISBN: 978-3-642-10665-1. DOI: 10.1007/978-3-642-10665-1_63.

[28] Jay Shah and Dushyant Dubaria. “Building Modern Clouds: Using Docker, Kubernetes & Google Cloud Platform”. In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC). Jan. 2019, pp. 0184–0189. DOI: 10.1109/CCWC.2019.8666479. URL: https://ieeexplore.ieee.org/abstract/document/8666479 (visited on 02/26/2025).

[29] Maarten van Steen and Andrew S. Tanenbaum. “A brief introduction to distributed systems”. In: *Computing* 98.10 (Oct. 1, 2016), pp. 967–1009. ISSN: 1436-5057. DOI: 10.1007/s00607-016-0508-7. URL: https://doi.org/10.1007/s00607-016-0508-7 (visited on 02/07/2025).

[30] *Stockbroker: Definition, Types, Job Requirements, and Salaries*. Investopedia. URL: https://www.investopedia.com/terms/s/stockbroker.asp (visited on 02/27/2025).

[31] Shanti Swaroop Moharana. *(PDF) Analysis of Load Balancers in Cloud Computing*. ResearchGate. URL: https://www.researchgate.net/publication/236521813_Analysis_of_Load_Balancers_in_Cloud_Computing (visited on 02/24/2025).

[32] Andrei Taranchenko. *Death By a Thousand Microservices*. URL: https://renegadeotter.com/2023/09/10/death-by-a-thousand-microservices.html (visited on 02/24/2025).

[33] *The Istio service mesh*. Istio. URL: https://istio.io/latest/about/service-mesh/ (visited on 03/14/2025).

[34] *What are Cloud Containers? - Cloud Containers Explained - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/what-is/cloud-containers/ (visited on 02/26/2025).

[35] *What is Amazon EKS? - Amazon EKS*. URL: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html (visited on 02/27/2025).

[36] *What is Amazon Relational Database Service (Amazon RDS)? - Amazon Relational Database Service*. URL: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html (visited on 02/27/2025).

[37] *What is Apache Kafka?* Confluent. URL: https://www.confluent.io/what-is-apache-kafka/ (visited on 02/24/2025).

[38] Bill Wilder. “Cloud Architecture Patterns”. In: *Cloud Architecture Patterns*. "O’Reilly Media, Inc.", 2012, pp. 3–4. ISBN: 978-1-4493-1977-9.

A PostgreSQL Configuration files

A.1 PostgreSQL Node 1

```
# DB Version: 14
# OS Type: linux
# DB Type: web
# Total Memory (RAM): 60 GB
# CPUs num: 7
# Connections num: 1600
# Data Storage: ssd

max_connections = 1600
shared_buffers = 15GB
effective_cache_size = 45GB
maintenance_work_mem = 2GB
checkpoint_completion_target = 0.9
wal_buffers = 16MB
default_statistics_target = 100
random_page_cost = 1.1
effective_io_concurrency = 200
work_mem = 30MB
huge_pages = try
min_wal_size = 1GB
max_wal_size = 4GB
max_worker_processes = 7
max_parallel_workers_per_gather = 4
max_parallel_workers = 7
max_parallel_maintenance_workers = 4
```

A.2 PostgreSQL Node 3

```
# DB Version: 14
# OS Type: linux
# DB Type: web
# Total Memory (RAM): 120 GB
# CPUs num: 15
# Connections num: 1600
# Data Storage: ssd

max_connections = 1600
shared_buffers = 30GB
effective_cache_size = 90GB
maintenance_work_mem = 2GB
checkpoint_completion_target = 0.9
wal_buffers = 16MB
default_statistics_target = 100
random_page_cost = 1.1
effective_io_concurrency = 200
work_mem = 30MB
huge_pages = try
min_wal_size = 1GB
max_wal_size = 4GB
```

```
max_worker_processes = 15
max_parallel_workers_per_gather = 4
max_parallel_workers = 15
max_parallel_maintenance_workers = 4
```

A.3 PostgreSQL Node 3

```
# DB Version: 14
# OS Type: linux
# DB Type: web
# Total Memory (RAM): 250 GB
# CPUs num: 31
# Connections num: 1600
# Data Storage: ssd
max_connections = 1600
shared_buffers = 64GB
effective_cache_size = 192GB
maintenance_work_mem = 2GB
checkpoint_completion_target = 0.9
wal_buffers = 16MB
default_statistics_target = 100
random_page_cost = 1.1
effective_io_concurrency = 200
work_mem = 30MB
huge_pages = try
min_wal_size = 1GB
max_wal_size = 4GB
max_worker_processes = 31
max_parallel_workers_per_gather = 4
max_parallel_workers = 31
max_parallel_maintenance_workers = 4
```