
# Master Thesis

**Master of Software Engineering (M.sc)**

**Department of Tech and Software**

**Major: Software Engineering (SE)**

## Topic:

**Enhancing microservice architecture resilience against transient faults with Saga Patterns for distributed transactions.**

Author: Amzi Maai Ham Ortega

Matrikel-Number: 31228627

First supervisor: Prof. Rand Kouatly

Second supervisor: Prof. Mohammed Nazeh Alimam

## Statutory Declaration:

I hereby declare that I have developed and written the enclosed Bachelor Thesis completely by myself and have not used sources or means without declaration in the text. I clearly marked and separately listed all the literature and all the other sources which I employed when producing this academic work, either literally or in content. I am aware that the violation of this regulation will lead to failure ofthe thesis.

Berlin,

.....

Place, Date 05.01.2024 (*Signature Maai Ham*)

**To my family, my friends, and God.**

**Thanks for all your support along the way.**

## Abstract

Many software projects can grow with time and become very difficult to work with. That is the reason Microservices architecture has gained popularity in the last couple of years since it enables the segregation of bigger complex application features into smaller pieces that can scale on their own. Many big companies like United Airlines, Uber, Google, and many more use this architecture since it enables easier development, deployment, and maintainability.

Many other benefits come as well, such as individual scaling of each microservice independently, reducing bottlenecks that affect performance, and managing resources better.

However, this architecture brings many other challenges. A very specific and common challenge is distributed transactions. Which are transactions that require different microservices actions to complete one single transaction for the user. Many patterns have emerged to tackle this issue, and each implementation brings different advantages and disadvantages. In this research, we focus on the Saga pattern, one of the most used and scalable for microservices architecture. We di

However, even having a pattern that solves distributed transactions, there is the imminent possibility of transient faults, which include network delays, service timeouts, server failures, and many others. And the possibility only rises with the amount of microservices.

In this research, we investigate ways to enhance the resilience of an application. This comprehends the ability to keep working despite transient events. We also look for consistency improvement, which comprehends the management of any incomplete transaction that could compromise any data integrity.

Finally, the research dives into a real-world scenario environment, where transient faults are injected in a controlled manner to see how the complete architecture resilience and consistency are affected.



Keywords
(Saga Patterns, microservice architecture, resilience, consistency, distributed transactions, data integrity, software architecture)

Table of Contents

**1 Chapter: Introduction ..... 11**

**1.1 Overview ..... 11**

**1.2 Objective ..... 12**

**1.3 Scope ..... 12**

**2 Chapter: Theoretical Background ..... 13**

**2.1 Literature Review ..... 13**

2.1.1 Pear-review articles and publications ..... 13
2.1.2 Books used for the research. ..... 14
2.1.3 Additional resources used in the research. ..... 15

**2.2 Research Questions ..... 16**

**2.3 Microservice Architecture overview and implications ..... 16**

**2.4 Distributed Transactions ..... 18**

2.4.1 Transaction Atomicity ..... 18
2.4.2 Two Phase-Commit ..... 18
2.4.3 Two-Phase Commit Shortfalls ..... 19
2.4.4 Saga Pattern ..... 20

**2.5 Resilience Patterns ..... 22**

2.5.1 Transient Faults ..... 22
2.5.2 Retry Pattern ..... 22
2.5.3 Circuit Breaker ..... 23
2.5.4 Timeout ..... 24
2.5.5 Correlation ID ..... 25
2.5.6 Fallback ..... 27
2.5.7 BulkHead ..... 28
2.5.8 Health Checks for Microservices ..... 28
2.5.9 Resilience Value in Distributed Services ..... 29

**2.6 Containerization ..... 30**

2.6.1 Docker ..... 31
2.6.2 Container Orchestration ..... 32
2.6.3 Kubernetes for Resilient Systems ..... 34

**2.7 Database Replication ..... 37**

2.7.1 MongoDb Replication ..... 37

**2.8 Chaos-Engineering ..... 40**

2.8.1 Linear and Nonlinear Systems ..... 40
2.8.2 Consistent Complexity ..... 41
2.8.3 Experimenting and Testing ..... 41
2.8.4 Verification versus Validation ..... 42
2.8.5 Chaos Engineering Principles ..... 42
2.8.6 Chaos Mesh ..... 43

**2.9 Message Brokers ..... 49**

2.9.1 RabbitMQ ..... 49

**2.10 System Under Test ..... 53**

2.10.1 NserviceBus ..... 56
2.10.2 NET ..... 58

**2.11 Metrics ..... 59**

2.11.1 Response Time ..... 59

2.11.2 Distributed Tracing ..... 60
2.11.3 Open Telemetry ..... 61
2.11.4 Jaeger ..... 62
2.11.5 Percentile ..... 62

**3 Chapter 3: Methodology ..... 63**

**3.1 Steady State Performance ..... 64**

**3.2 Hypothesis ..... 66**
3.2.1 Total outages in services ..... 66
3.2.2 Network outages ..... 67
3.2.3 Stress Degradation ..... 67

**3.3 Hypothesis validation and response time. ..... 67**

**4 Chapter: Results ..... 68**

**4.1 Pod or Service Failure Experiment ..... 68**
4.1.1 Pod Failure Experiment ..... 68
4.1.2 Pod Kill Experiment ..... 70
4.1.3 Third-Party Service Failure Experiment ..... 70

**4.2 Network Outage Experiment ..... 71**

**4.3 Stress Failures ..... 72**

**5 Chapter: Discussions ..... 74**

**5.1 Discussion of Service Outages ..... 74**
5.1.1 Container Failure Experiment ..... 74
5.1.2 Container kill experiment. ..... 74
5.1.3 Third-party Service Outage Experiment ..... 75
5.1.4 Experiments Hypothesis ..... 75

**5.2 Discussion of Network Outages ..... 76**
5.2.1 Experiments Hypothesis ..... 76

**5.3 Discussion of resource stress Outages ..... 76**
5.3.1 Experiments Hypothesis ..... 77

**5.4 Discussion of the Ideal High Resilience System ..... 77**
5.4.1 Enhancing the System with Persistence. ..... 78
5.4.2 Enhancing the System with Retry Pattern. ..... 78
5.4.3 Enhancing System with Circuit Breaker. ..... 79
5.4.4 Enhancing System with Automatic Scaling. ..... 79
5.4.5 Enhanced System Results ..... 80

**6 Chapter: Implication and Future Results ..... 82**

**6.1 Conclusion ..... 82**

**6.2 Limitation of Chaos Engineering Experiments ..... 82**

**6.3 Experiments Improvements ..... 83**

**6.4 Future Research ..... 83**

**Bibliography ..... 84**

**Appendix ..... 91**

**List of Figures**

Figure 2.1 Two-Phase commit pattern. Source: [33]. ..... 19
Figure 2.2 Two-Phase commit pattern, Abort. Source: [13]. ..... 19
Figure 2.3 Two-Phase commit pattern, Deadlock. Source: [35]. ..... 20
Figure 2.4 Saga pattern diagram. Source: [34]. ..... 20
Figure 2.5 Saga choreography pattern diagram. Source: [15]. ..... 21
Figure 2.6 Retry pattern diagram. Source: [37]. ..... 23
Figure 2.7: Circuit Breaker pattern diagram. Source: [20]. ..... 24
Figure 2.8 Timeout pattern diagram. Source: [38]. ..... 25
Figure 2.9 Correlation Id. Source: [40]. ..... 26
Figure 2.10 Transaction tracing diagram. Source: [24]. ..... 26
Figure 2.11 Fallback diagram. Source: [43]. ..... 27
Figure 2.12 Fallback diagram. Source: [29]. ..... 28
Figure 2.13 Transaction tracing diagram. Source: [15]. ..... 29
Figure 2.14 Isolation testing issue. Source: [16]. ..... 30
Figure 2.15 Containerization compared with traditional virtualization. Source: [49]. ..... 31
Figure 2.16 Deployment declarative statement Kubernetes. Source: [55] ..... 33
Figure 2.17 Rolling Update Kubernetes Diagram. Source: [56]. ..... 34
Figure 2.18 Resource model for Kubernetes Gateway API. Source: [66]. ..... 36
Figure 2.19 Rolling Update Kubernetes Diagram. Source: [71]. ..... 37
Figure 2.20 MongoDb replication model. Source: [77]. ..... 38
Figure 2.21 MongoDb replication read operations. Source: [63]. ..... 39
Figure 2.22 MongoDb replication automatic failover. Source: [63]. ..... 39
Figure 2.23 Simple and Complex Systems. Source: [21]. ..... 41
Figure 2.24 Chaos Mesh architecture overview: [83]. ..... 44
Figure 2.25 Chaos Mesh Pod-Failure example: [85]. ..... 45
Figure 2.26 Chaos Mesh Pod-Kill example. Source: [85] ..... 45
Figure 2.27 Chaos Mesh Pod-Container-Kill example. Source: [77] ..... 45
Figure 2.28 Chaos Mesh Network emulation example. Source: [86] ..... 46
Figure 2.29 Chaos Mesh Network Partition example. Source: [78] ..... 47
Figure 2.30 Chaos Mesh Network Partition example. Source: [78] ..... 47
Figure 2.31 Chaos Mesh Stress simulation example. Source: [87] ..... 48

Figure 2.32 Chaos Mesh AWS EC2 failure simulation example. Source: [79]..... 48
Figure 2.33 Chaos Mesh AWS volume failure simulation example. Source: [79]..... 49
Figure 2.34 Exchange element in RabbitMQ. Source: [19]..... 51
Figure 2.35 RabbitMQ Work Queue. Source: [94]..... 51
Figure 2.36 Exchange element in RabbitMQ. Source: [85]..... 52
Figure 2.37 Loosely coupled architecture with RabbitMQ. Source: [85]..... 53
Figure 2.38 Workflow for E-commerce distributed transaction. Source: own..... 54
Figure 2.39 Architecture for e-commerce distributed transaction. Source: own..... 55
Figure 2.40 Saga transient failure without retry pattern. Source: [20] ..... 57
Figure 2.41 Saga transient failure with retry pattern. Source: [89]..... 57
Figure 2.42 Saga data state with NServiceBus persistence. Source: own ..... 58
Figure 2.43 Resilience metric for a power system. Source: [13]..... 59
Figure 2.44 Waterfall diagram of a distributed transaction. Source: [105] ..... 60
Figure 2.45 Diagram of OpenTelemetry architecture. Source: [103]..... 61
Figure 2.46 Trace Detail View Screenshot. Source: [108] ..... 62
Figure 2.47 Average and Percentile Source: [103]..... 63
Figure 3.1 Trace timeline. Source: own..... 64
Figure 3.2 Trace flame graph. Source: own..... 64
Figure 3.3 Plot graph of response time. Source: own ..... 65
Figure 4.1 Pod Failure Experiment Definition. Source: own ..... 68
Figure 4.2 Pod Failure Experiment Status. Source: own ..... 69
Figure 4.3 Detailed Trace for Pod Failure Experiment. Source: own..... 69
Figure 4.4 Pod Kill Experiment Definition. Source: own..... 70
Figure 4.5 Pod Kill Experiment Status. Source: own ..... 70
Figure 4.6 Third-party or unhandled exceptions experiment. Source: own..... 71
Figure 4.7 Detailed Trace for Third-Party Failure Experiment. Source: own ..... 71
Figure 4.8 Network experiment definition. Source: own..... 71
Figure 4.9 Network experiment result. Source: own ..... 72
Figure 4.10 Stress experiment result. Source: own..... 72
Figure 4.11 Stress experiment CPU monitor. Source: own ..... 73
Figure 4.12 Stress experiment Trace detail. Source: own..... 73
Figure 5.1 Saga data state persistence in MongoDB. Source: own ..... 78
Figure 5.2 Retry pattern in NServiceBus. Source: own..... 79

Figure 5.3 Replica set of notification Service in Kubernetes. Source: own ..... 80
Figure 5.4 Enhanced system trace detail with pod failure experiment. Source: own ..... 80
Figure 5.5 Enhanced system trace detail with third-party outage experiment. Source: own ..... 81
Figure 5.6 Enhanced system trace detail with network outage experiment. Source: own ..... 82

Table of tables:

Table 1 Base Response Time Source: Own ..... 66
Table 2 Response Time Percentile ..... 66
Table 3 Results of experiments before and after enhancements ..... 82

# 1 Chapter: Introduction

Microservice architecture has gained popularity for building scalable, modular, and flexible systems. This architecture enables the independent scalability, of a module, meaning that resources are managed better, and available for reuse. It also brings the ability for easier development, technology heterogeneity, and more importantly the isolation of [1].

Moreover, the adoption of microservice architecture is not limited to specific industries. Many big companies, not related to each other like fintech, eHealth, and hospitality such as Netflix, Amazon, and Uber, have adopted this architecture on a big scale [2].

But even though this architecture solves growth issues and brings benefits, it also comes with some challenges related to reliability and data integrity for distributed transactions. In the face of this challenge, design patterns like the Saga Pattern have emerged and gained traction [3].

The saga design pattern decomposes distributed transactions into many local transactions. This can be done in different ways, which is why this design pattern can have different implementations. However, this design pattern is susceptible to different transient faults that are present in real-world environments. These include network congestion faults, scalability issues, high latency, server overload conditions, CPU hog, memory hog, disk hog, node crash faults, and others [4].

This research explores different ways to enhance the Saga pattern's reliability and data integrity against these transient faults.

## 1.1 Overview

Even though monolith architectures, which is the counterpart of the microservices architecture, do not present the challenges of distributed transactions, they do present a single point of failure [5]. This means that an application's complete availability can be affected by a single instance of a transient fault.

The microservice architecture avoids this point of a single failure. As a result, any damage caused by a transient fault is reduced to a single microservice. And even though distributed transactions need the availability of all microservices, some transactions could still be operational. This only means that the resilience of a microservice architecture is just as strong as its weakest microservice.

Knowing that each of our microservices is exposed to its transient faults, we can say that this decentralized architecture can be a double-edged sword. But with already-known practices like resiliency patterns, we can get the best of each pattern; a good resilient system capable of scaling with no problem [6].

## 1.2 Objective

1. Identify and analyze the common challenges and faults that affect the system resilience and data integrity of a microservice architecture using the Saga pattern.
2. Investigate resilience design patterns and strategies and how they can be applied to the Saga patterns in microservice to enhance the system resilience and integrity against real-world transient errors.
3. Analyze and compare how these resilience patterns and strategies benefit the saga pattern in a real-world scenario by the observation of environmental experiments with deliberately faulted injections.

## 1.3 Scope

The scope of this research includes the investigation of common challenges of distributed transactions of microservice architecture and how these are tackled by the saga pattern architecture. The study aims to research ways to enhance the system's reliability and data integrity against common transient faults found in real-world environments. Finally, testing of these enhancements is tested with deliberately injected faults in a controlled environment to confirm our hypothesis and to make recommendations.

The scope of this research also includes:

- Identification and analysis of distributed transactions challenges.
- Analysis of the saga pattern for distributed transactions.
- Evaluation and comparison of different saga pattern implementations.
- Identification and evaluation of resilience patterns.
- Evaluation of the application of resilience patterns in the saga pattern.
- Identification of potential points of failure in the architecture.
- The development of a testing framework capable of injection of deliberately common real-life transient faults.

The scope of this study does not include:

- Security concerns about the architecture, including the communication between microservices.
- Resource and cost management of the architecture.
- Scalability of the architecture.

# 2 Chapter: Theoretical Background

## 2.1 Literature Review

### 2.1.1 Pear-review articles and publications

With the rise of the popularity of microservices during the last few years, many studies have been published. The following studies included, are about microservices, or related to distributed transactions. Other studies such as resilience and cloud failures are considered to research how to enhance the resilience of the distributed transactions with saga patterns.

This study [7] researches how distributed systems have become common in today's daily activities, due to better management and distribution of resources, and how these can lead to the unavailability of services due to multiple failures. The study addresses these failures with common practices such as high availability.

This study [8] researches how cloud providers have taken a big part of responsibility in availability and how service level agreements can help to mitigate expected failures in the cloud. The research aids any involved party in cloud services to design better reliable fault-tolerant systems to enhance the availability and unscheduled downtimes in the system. It archives this by creating machine learning models, based on 2 years data of from the National Energy Research Scientific Computing Center, which predicts hardware failures of cloud providers in real-time.

The study [9] conducted by Rudrabhatla brings valuable data about NoSQL databases in the comparison of choreography and orchestration in microservice architecture. Even though a two-phase commit is not supported by NoSQL databases, the research elucidates the challenges and presents fact-based recommendations to the saga pattern and NoSQL databases.

This study [10] proposes a diverse approach to microservices to enhance resilience and availability. The study shows an enhancement in overall reliability by a proposed model of reduced service dependence, avoiding overreliance on microservices through decentralized deployment.

This study [11] proposes a formal approach in software development specifications for tolerating transient faults that can spread across different systems (microservices). The study shows in an avionics software study case that even these are short-lived errors, that can affect the overall system.

This study [12] investigates cyber-physical systems that incorporate complex engineering systems such as water distribution networks, healthcare systems, manufacturing, and so on. The study applies the principle of Chaos Engineering discipline to experiment and find resilience-enhancement techniques in a real-time operation. The study demonstrates the benefits of such practices on system resilience.

This study [13] investigates the resilience enhancement for a power grid system. Even though the study is not related to software, it is related to resilience enhancement

and gives precedence in the selection of metrics to measure the resilience of a system. This study is used as a guideline for the selection of the resilience enhancement metric for distributed systems.

Finally, this study [14] evaluates the saga pattern for the microservices architecture. The study evaluates the technologies for implementing the saga pattern and provides a general insight into what should be kept in mind while an implementation of this pattern is applied to microservices architecture.

While these studies provide valuable insights about microservices, distributed transactions, transient faults, and resilience enhancement, there is still a need for a comprehensive evaluation of how resilience can be enhanced in saga patterns for distributed transactions, by measuring it and testing it. This research aim is to address the gap between distributed transaction research and resilience enhancement for software systems. The research plans to provide valuable insights to developers, and software architects on how to manage unexpected transient or systematic failures in distributed transactions with microservice architecture, by learning through experimentation for the enhancement of them.

### 2.1.2 Books used for the research.

Several books explaining different related topics to the topic of the research were used. The book “Spring Microservices in Action” by John Carnell [15], provides insight into the microservices architecture, its benefits, and its implementation. It gives an introduction to the cloud and how the microservices are usually deployed into the cloud. More importantly, the book gives insights for recognizing the proper needs for when a microservice architecture is useful and when it is not, since it defines also the complexity and drawbacks of this architecture. The book also gives resilience patterns for microservice architecture and delves into how failures can propagate into other services.

The book “Release It! Second Edition” by Michael T. Nygard [16] delves into how to design and deploy proper production-ready software. The book uses a case study that shows how exceptions can cause outages in a system, their consequences, and how they could have been prevented. After this, the book delves into failures, and chain reactions of these in a microservice architecture. It discusses stability patterns and antipatterns, commonly seen in software design. Finally, the book also dives into chaos engineering and why breaking them can help to make them better. In other words, running experiments to learn about systems.

The book “The Docker Book: Containerization Is the New Virtualization” by James Turnbull [17] provides wide information about how software virtualization has become a new standard in software deployment. The book gives a detailed guide about software containerization without skipping any detail about virtualization, containers, docker, and related tools.

The book “Docker Orchestration” by Randal Smith delves into the necessity of docker orchestration due to the nature of the high amount of services in the microservice

architecture. It gives detailed information about how multiple container applications should be managed, regarding their network setup and data volumes, while keeping security in mind. The book mentions orchestration tools briefly, such as Kubernetes which is explained in more depth in the book “Kubernetes Up & Running” by Brendan Burns, Joe Beda, and Kelsey Hightower [18].

This later book gives a detailed explanation of how to use Kubernetes efficiently by taking advantage of its tools. It defines the value of velocity provided by docker orchestration and how self-healing systems can be developed with an infrastructure controlled by Kubernetes. More important the book explains how horizontal scaling can help the system's availability and resilience, against infrastructure failures and how stateful services such as databases present different challenges such as data persistence, data integrity, and synchronization.

For the crucial communication of microservices, the book “RabbitMQ in-depth” by Roy Gavin [19] introduces the open-source, well-known, and reliable message broker pattern, using RabbitMQ. The book explains how software, especially dependent systems can benefit from decoupled architectures and how this can be achieved by communication with message brokers.

For the saga implementation, the NserviceBus framework is exposed by the book “Learning NserviceBus Second Edition” by David Boike [20]. The book starts by explaining how long-running processes are constant due to business processes, and how these can be replaced by sagas. The long-running processes are identical processes of distributed transactions according to the book, which can be addressed by the saga implementation of the NserviceBus framework. The book not only extends the saga discussion but also on the benefits of the native features of the framework such as the resilience patterns timeout patterns, retry, correlation identifiers, and persistence.

In matters of resilience, the book “Chaos Engineering” by Casey Rosenthal and Nora Jones [21] gives broad information about the antecedents and origins of this discipline and how it has evolved into a well-accepted methodology to test a system's resilience. The book explains which systems should be targeted by this discipline and how complex systems can be identified. It also defines the difference between tests against experiments and verifications and validation.

### 2.1.3 Additional resources used in the research.

Even though research papers and formal books were used as the main resources for this study, documentation of frameworks and technologies was also used since many of the features described in the books are deprecated or updated.

Other resources such as journals and conference papers were also used by their close relationship to aspects of this research. These journal papers, even though not highly related to the main aspects of this research, provide valuable data.

The journal paper [22] about the state of the art of messaging design for distributed computing systems, gives an overall introduction to messaging for microservice architecture and message-oriented middleware.

The journal paper [23] about tracing in distributed transactions and microservice architecture gives insight into how the allocation of information between different microservices for a specifically distributed transaction can be identified with correlation IDs.

## 2.2 Research Questions

1. How do transient faults affect distributed transactions' resilience and data integrity when using a saga pattern for microservices architecture?

2. What best practices or strategies can enhance resilience and data integrity against common transient faults?

3. What effectiveness and applicability have these practices and strategies against transient faults in real-world scenarios?

## 2.3 Microservice Architecture overview and implications

*Monolithic Architecture*

Monolithic architecture refers to a style or architectural software infrastructure, where the application's components are combined into a program. This architecture is characterized by having just one unit, which becomes difficult to maintain as the application functions increase over time [24].

*Microservice Architecture*

Microservice architecture refers to a style or architectural software infrastructure, where the application is divided into a subset of smaller, independent services running able to communicate with each other [25].

*Software resilience*

Software resilience refers to the ability of the software system to withstand various disturbances and maintain a desired level of service against these. It also comprehends the adaptability of the system to different levels of disturbances over time [26].

*Distributed Transactions*

Distributed transactions refer to a set of operations that mutate state needed to be executed atomically. This ensures that whether all operations are executed successfully or not, to maintain data consistently and integrity [27].

*Saga Pattern*

The saga pattern refers to the approach mostly used for distributed transactions. The approach breaks down multiple-step transactions into smaller operations that can be undone by compensation action in case the system has to cancel, a user decision, or a system major error [28].

*Transaction Atomicity*

Transaction atomicity refers to a transaction that should be either completed successfully or unsuccessfully by either committing or aborting the transaction. When it is unsuccessfully, the transaction should be as if it has never been started, the database should not be affected in any way [29].

*Message Broker*

Message broker refers to the component that acts in the middle of others and enables them to communicate between them. These components are usually a pair of publishers and subscribers [30]. Furthermore, the message broker ensures the scalability and fault tolerance of the application load [22].

*Point of failure*

Point of failure refers to a component, process, or condition in the infrastructure that can lead to failure or a complete malfunction of the system [15].

*Load Balancing*

Load balancing refers to the practice of distributing requests across a pool of resources to serve them in the shortest amount of time, with the main objective of using all the resources evenly [16].

*Docker*

“Docker is an open-source engine that automates the deployment of applications into containers. It was written by the team at Docker, Inc (formerly dotCloud Inc, an early player in the Platform-as-a-Service (PAAS) market), and released by them under the Apache 2.0 license.” [17].

*Infrastructure as a Code*

Infrastructure as a Code refers to the practice that involves defining and managing the infrastructure of an application as machine-readable scripts or code. This approach applies software development principles to infrastructure code, to automate the provisioning and maintenance of infrastructure resources [31].

## 2.4 Distributed Transactions

The microservice architecture manages transactions in a different way than ordinary monolithic architecture. These transactions update data on two or more databases or computer systems.

Since monolithic architecture counts with only one broker against the database

### 2.4.1 Transaction Atomicity

The challenge with distributed transactions is found in transaction Atomicity (previously defined). The microservices implementation must be robust to withstand multiple failures to keep transaction atomicity. Any failure in any service could result in the compromise of data integrity.

Nonmicroservice architecture such as the monolithic approach, does not suffer as badly from this challenge. Even though atomicity is attained by different approaches, such as Unit of Work, the probability of failure is lessened since only one point of failure exists.

### 2.4.2 Two Phase-Commit

The Two-Phase commit protocol approach tries to solve the distributed transactions challenge by making sure that microservices commit a transaction if and only if all commit successfully [32].

As the name hints, the Two-Phase commit protocol consists of two phases to ensure the atomicity of distributed transactions. The first phase consists of a voting or preparation phase, while the second consists of a decision or commitment phase. In both phases, a coordinator is required for communication against the different microservices and decision-making [33].

During the first voting phase, the microservices check if their own desired transaction, which is part of the complete distributed transaction, is possible. If the transaction is possible, the object to update is prepared and blocked so that no other service or different transaction can update the record.

When the microservices report back to the coordinator that the first phase is complete, and only if all reports are successful, then the coordinator follows with the decision phase and orders the microservices to commit their previously prepared data.

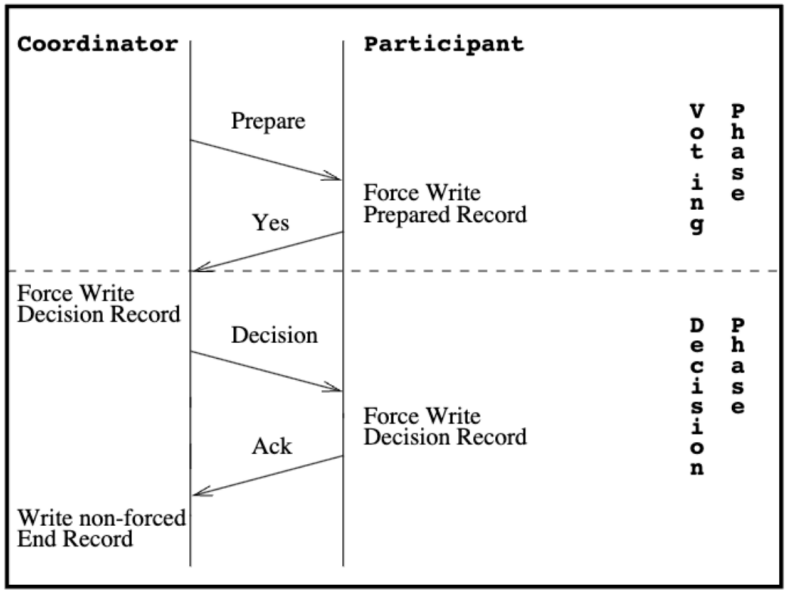



Figure 2.1 Two-Phase commit pattern. Source: [33].

When a microservice reports back to the coordinator that is not able to continue in the first phase, the coordinator is forced to cancel the complete distributed transaction and communicates all microservices to abort their previously prepared data.

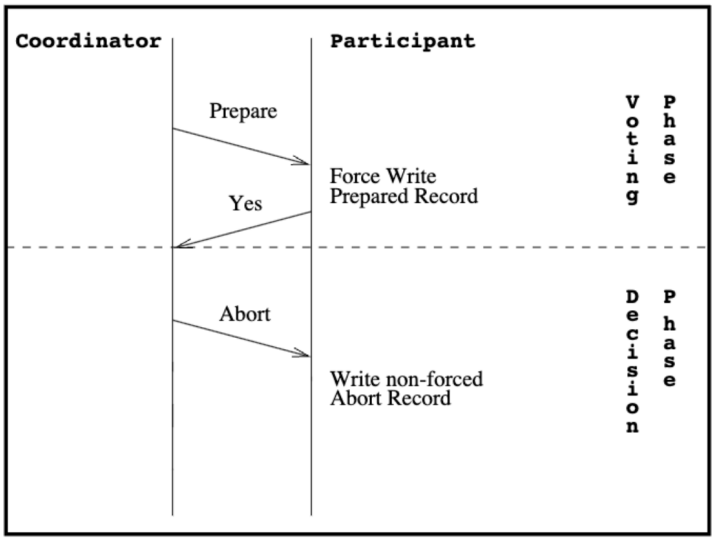



Figure 2.2 Two-Phase commit pattern, Abort. Source: [13].

### 2.4.3 Two-Phase Commit Shortfalls

#### 2.4.3.1 NoSQL Databases

The two-phase commit pattern relies on the ability of the database to commit changes in a transaction-wise manner. Unfortunately, not all of the NoSQL database engines support a transactional model [34].

#### 2.4.3.2 DeadLocks

Deadlocks are a common challenge the two-phase commit presents. This can happen when different microservices block one specific record in the first phase, or when they block records dependent on each other's transactions [35].

| Time | T1 | T2 |
|---|---|---|
| $t_0$ | BEGIN TRANSACTION | BEGIN TRANSACTION |
| $t_1$ | Get Write Lock for A (successful) | Get Write Lock for B (successful) |
| $t_2$ | Update A = A + 1 ( A = 2 ) | Update B = B * 2 ( B = 2 ) |
| $t_3$ | Get Write Lock for B (wait until $t_5$) | Get Write Lock for A (wait until $t_5$) |
| $t_4$ | Update B = B + 1 ( B = 3 ) | Update A = A * 2 ( A = 4 ) |
| $t_5$ | UNLOCK A | UNLOCK B |
| $t_6$ | UNLOCK B | UNLOCK A |
| $t_7$ | COMMIT TRANSACTION | COMMIT TRANSACTION |
| | | |

Figure 2.3 Two-Phase commit pattern, Deadlock. Source: [35].

### 2.4.4 Saga Pattern

The saga pattern provides a different solution to the distributed transactions challenge, by making each local transaction (a transaction step required to complete the main transaction) report back with a message, that when is successful triggers the next local transaction. In the same manner, if one local transaction in the series of steps is unsuccessful, a saga executes the required compensation steps to roll back any committed transaction so far [34].

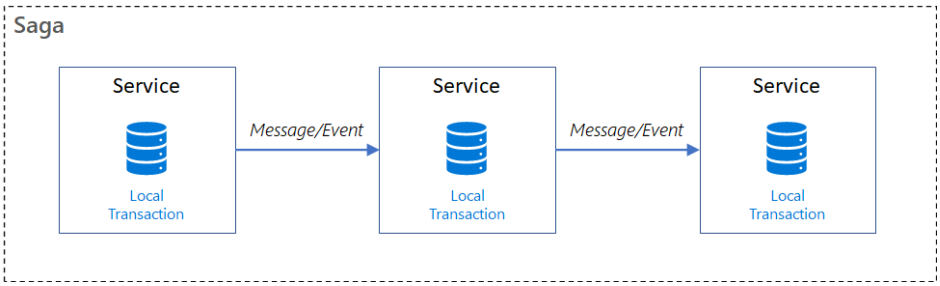



Figure 2.4 Saga pattern diagram. Source: [34].

#### 2.4.4.1 Orchestration

In the orchestration approach of the saga pattern, there is a dedicated service, called the orchestrator, that is responsible for calling all other services in an orderly and controlled manner. The orchestrator does this until the state is complete and reports back to the client to request that his transaction be complete. The orchestrator is also responsible for taking corrective actions if any of the local transactions fail [9].

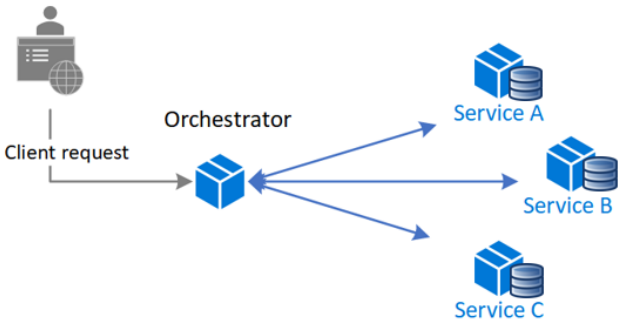



Figure 2.5: Saga orchestration pattern diagram. Source: [15].

#### 2.4.4.2 Choreography

The alternative implementation of the saga pattern is choreography, and as the name implies, this implementation does not rely on a single responsible service but the responsibility is shared among the services [9].

When one distributed transaction requires multiple microservices, each of these is responsible for publishing the result message to a shared message broker, meanwhile, the other microservices react to these messages as the continuation of the major distributed transaction or the compensation of any failed local transition.

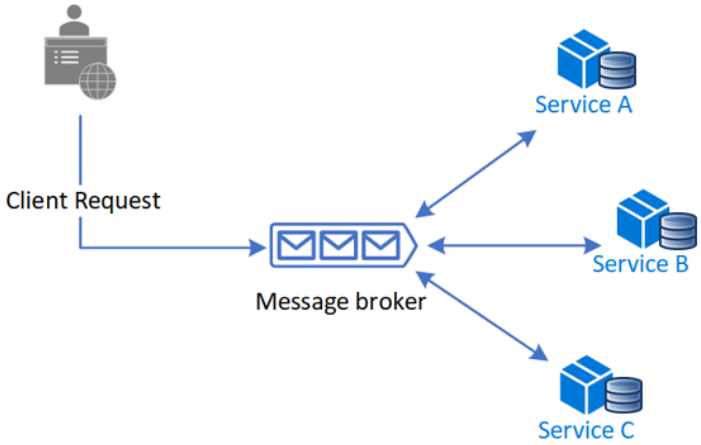



Figure 2.5 Saga choreography pattern diagram. Source: [15].

#### 2.4.4.3 Orchestration and Choreography

Choreography can introduce cyclical dependencies, causing a deadlock between microservices, because of the nature of one service requirement of being aware of the other's events, and how to handle its success and its failure. That is why this implementation is best used for low-complexity scenarios since it does not escalate well for big and complex [34].

On the other Orchestration can scale better for complex transactions, since microservices do not need to know about others' transactions and states, but since the flow is orchestrated from one service, a single point of failure is introduced [34].

#### 2.4.4.4 Single Point of Failure

In distributed systems, the presence of multiple points of failure can significantly impact the overall system availability and reliability. When strategies to improve the reliability of a system, like high redundancy and replication, are the same for all points of failure, it can be easier to implement them once, than multiple times [7].

Orchestration represents a single point of failure, and while choreography could advantage of the decentralized workflow, the research aims to improve resilience and no availability. Moreover, the research objective to enhance resilience is easier tested and measured in a single point of failure.

## 2.5 Resilience Patterns

### 2.5.1 Transient Faults

Managing transient faults in the cloud requires robust failure-managing mechanisms to ensure the availability and resilience of the services. Even though transient faults can be short-lived, and typically correct themselves after a brief period, they could affect service level agreements which affect the user [36].

In cloud-native environments, transient faults are not uncommon. Even though cloud providers commonly promise the availability of 99.9% of the time, transient errors can be found outside the availability scope and be issues of poor resource management by the user. This is why even though an application is unit-tested, integrated, and even end-to-end tested, it might encounter faults that the application was never developed for.

Some of the following patterns are commonly implemented to enhance a system's resilience against these types of unforeseen transient faults.

### 2.5.2 Retry Pattern

The retry pattern tries to enhance the system's resilience by retrying failed requests a defined number of times, with an incremental waiting time. This incremental waiting time gives the service a better opportunity to handle the following request correctly [37].

The retry pattern needs to be applied only in some cases since it will not help the resilience always. It can even deteriorate the system's performance if not used properly.

In requests, where failure is sure to happen again, for example, authentication error, the pattern would not add any benefit. However, when the failure is uncommon, and there is not a well-known reason for the next request to fail again, the pattern should be used.

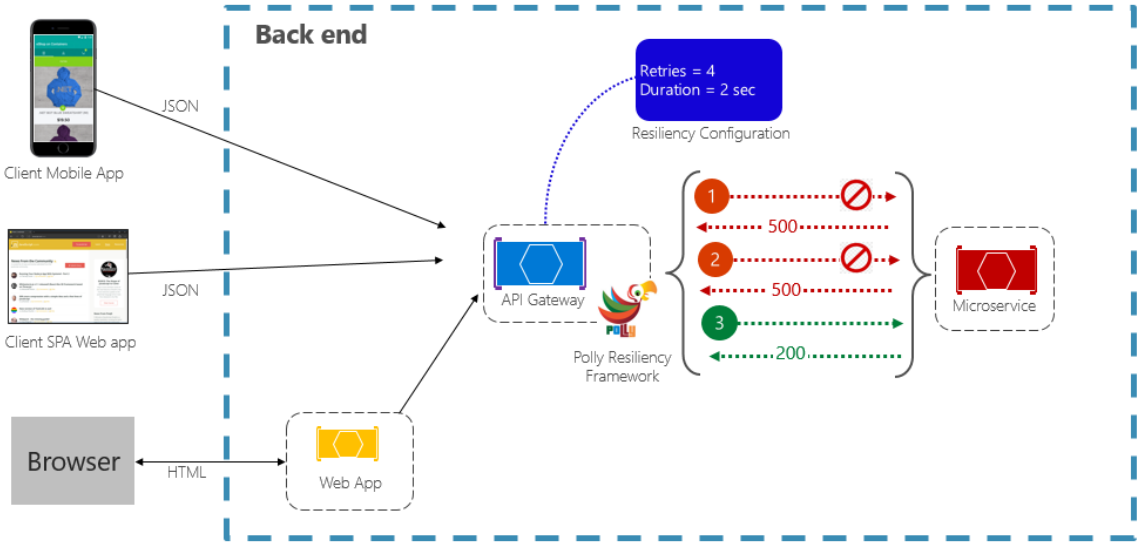



Figure 2.6 Retry pattern diagram. Source: [37].

In the previous diagram, the API gateway consumes a microservice, but it receives a failed response. The API gateway retries a second and third time with an exponentially growing waiting time on each request. Finally, the microservice sends a successful response and the API gateway can respond to the client.

### 2.5.3 Circuit Breaker

The circuit breaker pattern is inspired by the circuit breaker solution for poor-quality wiring in houses that make them catch fire. The idea of making an electrical breaker fail (subsystem) without burning the house (entire system) is the same in software. Instead of burning a house, the retrying of a request in an unavailable service could flood the message bus, and database connections and introduce errors to unrelated parts of the system [16].

In the next image, a circuit breaker pattern is applied to the already existing retry pattern. After a certain number of failed requests (100 in this example), the system decides not to retry again and open the circuit. After a determined amount of time, the system closes again, continuing the operation as usual.

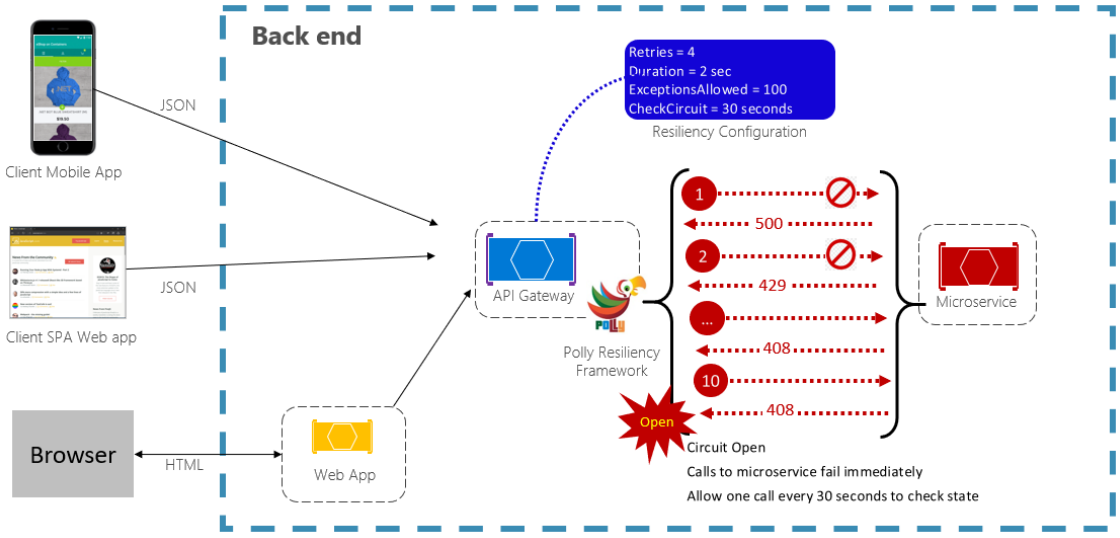



Figure 2.7: Circuit Breaker pattern diagram. Source: [20].

When a circuit should be opened and closed back again, is defined by different implementations. To open a circuit, the criteria could be the number of failed requests, while more complex criteria could be the frequency of these. Another strategy could also include an API health check, to know when to close the circuit again [16].

Any incoming requests during the open state of a circuit breaker pattern can be managed differently depending on the desired strategy. The system can fail early instead and could return a fallback value, or a cached value. It can have a secondary service or could queue the request for later retry when the circuit becomes closed [16].

### 2.5.4 Timeout

Since we know that distributed transactions are the subset of local transactions by different services, the major distributed transaction can be affected greatly by any service in the chain of the workflow.

We know that any resource is not unlimited, and therefore can always be exhausted. Distributed services have more resources dependent than monolith services. These resources can be network, computational, and provider-sided, meaning that there is a high probability of encountering requests that take more time than expected, negatively affecting the system's expected way of working [16]

.

The timeout pattern gracefully cancels any request that is not finished in the expected time. This pattern returns a timeout message, a success, or a note stating that the request was added to a queue for a later retry or processing. This pattern helps with the user experience since it's better to give early feedback instead of an undefined waiting time. It also helps by freeing services, instead of keeping resources busy due to an unrelated issue system [16].

The timeout pattern can also be synergetic used with the circuit breaker pattern. The historical timeout events can be used as a trigger to open any circuit breaker and

fail early. The system's robustness can also be enhanced with a store and retry strategy for these timeout requests by adding a retry queue [16].

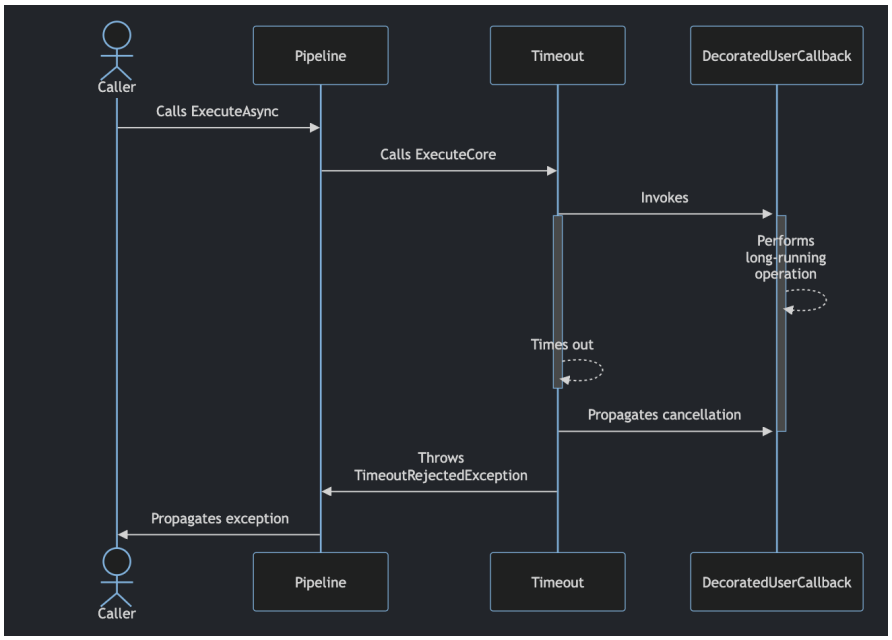



Figure 2.8 Timeout pattern diagram. Source: [38].

In the previous diagram, we can see the abstract diagram of how a timeout strategy can be implemented in an Orchestration saga pattern. Here the caller starts a distributed transaction, and in the timeout layer, which should be the orchestrator, a timeout timer starts as soon as we start a request to a microservice.

Should the microservice take more than the expected defined time, the timer would trigger a cancellation of the request, propagating the cancelation, freeing the allocated resources, and returning a response depending on the strategy of management of timeout messages.

### 2.5.5 Correlation ID

The correlation ID is how a service ties a reply to a request. In the saga pattern, many requests of distributed transactions can happen at the same time. This means that the orchestrator needs to differentiate what instance of the request he is getting responses from [39].

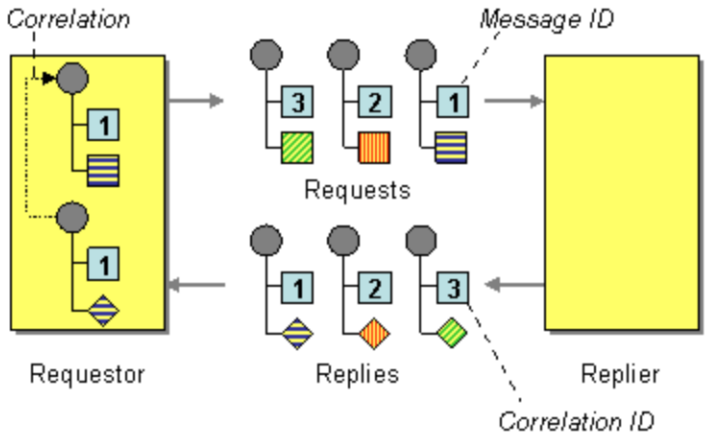



Figure 2.9 Correlation Id. Source: [40].

In the previous image, the requestor (orchestrator in the saga pattern) sends three requests with a unique correlation ID for each to the same destination. When the replier responds, each reply has the same correlation ID, which the requestor uses to match each major distributed transaction.

Distributed tracing enables the monitoring and analysis of a single distributed transaction request across different instances of different microservices. This enables easier debugging by tying logs from different microservices by a unique identifier that is injected in the orchestrator [23].

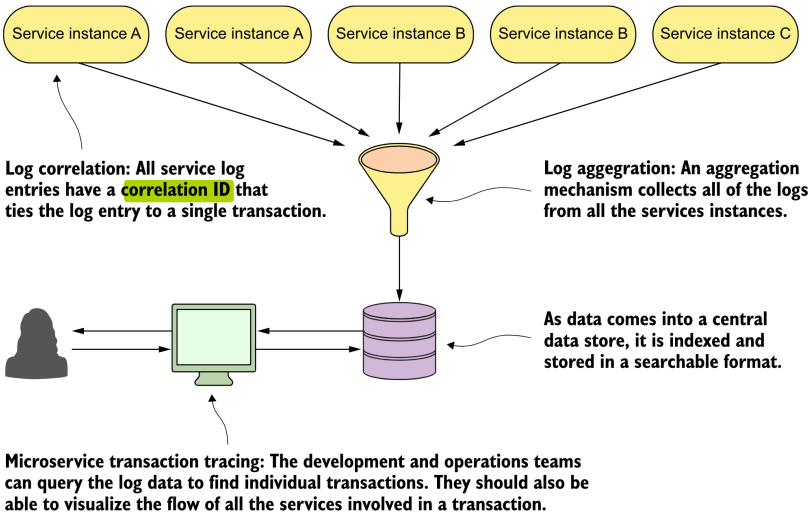



Figure 2.10 Transaction tracing diagram. Source: [24].

In the previous example, each microservice can have multiple instances incrementing the number of logs. With all microservices logs having a correlation ID, filtering after log aggregation is easier, therefore connecting all logs related to a single distributed transaction request.

Even though distributed tracing does not enhance resilience directly, it can give observability such as identifying bottlenecks, errors, and user-specific actions that lead

to errors, which enhance resilience indirectly [41]. In addition, system transparency is a good predictor if it will survive since it cannot be tuned or optimized for enhancing reliability and resilience when the developers do not know how it works, how it fails, and when and where it fails for distributed systems [16].

### 2.5.6 Fallback

The fallback pattern manages failed requests by filling with a default value instead. In this manner, if a microservice is unavailable, a different microservice can be requested to have alternative data, instead of an error [42].

For a transaction microservice, the fallback is applied differently. When a transaction fails, the fallback pattern can be applied for a later retry of a failed transaction, if no other step is dependent on this. This strategy of the fallback pattern must be examined closely if it solves rather than introducing new failure points [15].

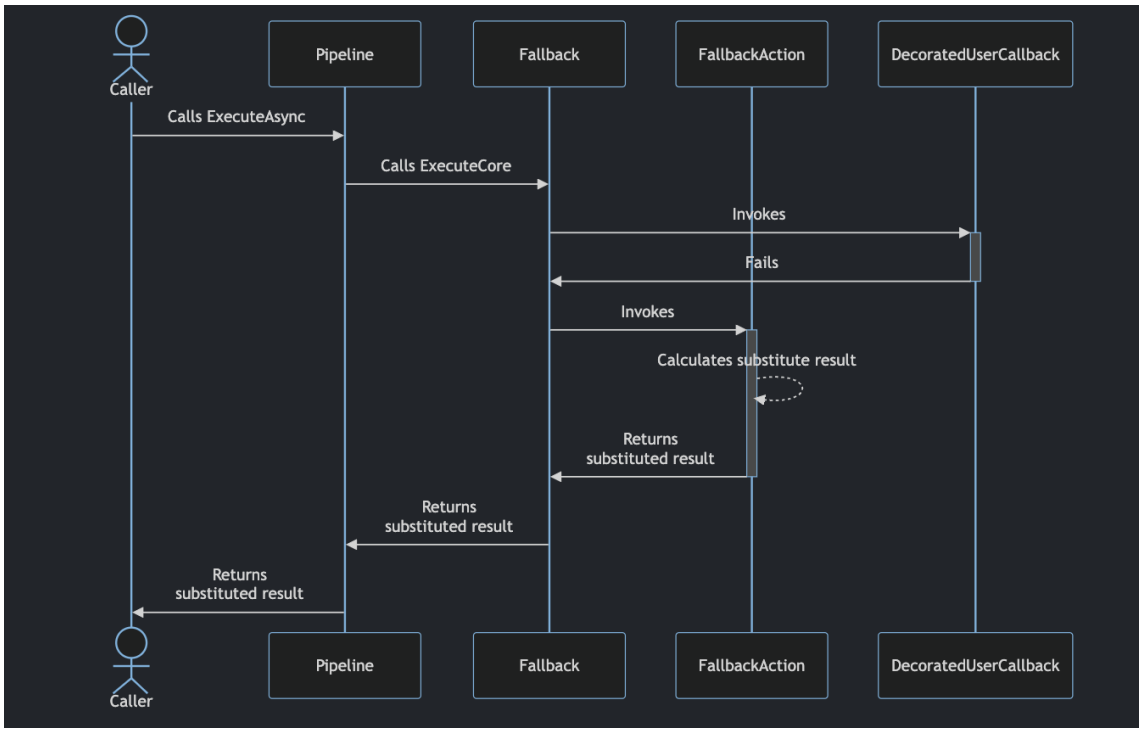



Figure 2.11 Fallback diagram. Source: [43].

The previous abstract diagram shows, how the fallback action triggers a substitute result when the primary service fails. The request returns a value, even though it can come from a different dataset or different calculation, but it is better than an error.

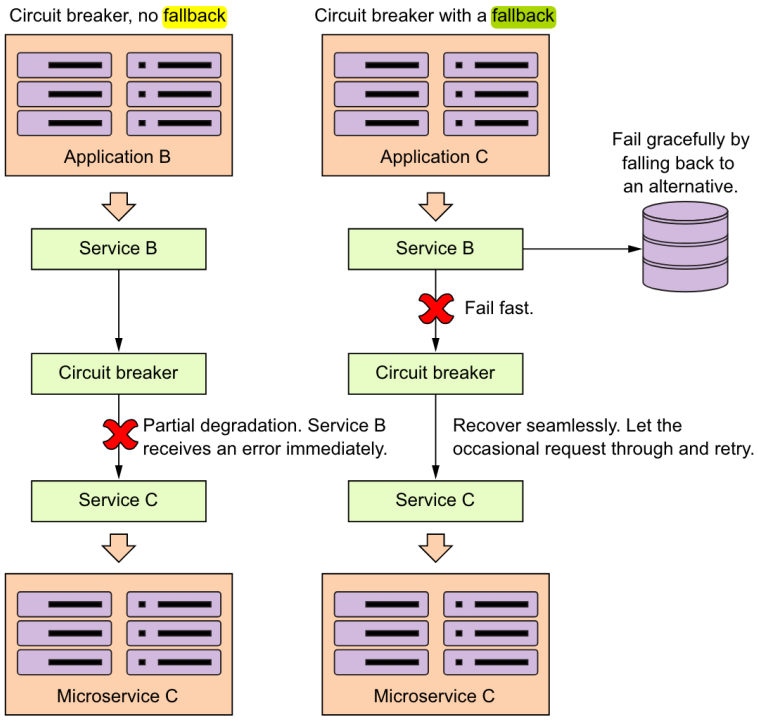



Figure 2.12 Fallback diagram. Source: [29].

Most of the time, the fallback pattern is used in conjunction with the circuit breaker pattern. That is because, a circuit breaker that is not forwarding the request to the intended service (open), can use a fallback service meanwhile the first intended microservice is reestablished [15].

### 2.5.7 BulkHead

The bulkhead partition pattern implies the partition of a system into parts independent of each other, enforcing the principle of damage containment as a ship [16].

### 2.5.8 Health Checks for Microservices

A proactive way to increase a system's resilience is the monitoring of the health of a microservice and it can be assessed by their need's status. These can be runtime dependencies, database connections, resource availability, and many others. A health check endpoint can be used to monitor each microservice by which the orchestrator can manage their circuit breakers, fallback, and retry pattern strategies [44].

For health checks, there must be a monitoring service, which is what watchdogs are for. This service checks and reports the health of microservices. It is also a great place to host code to remediate actions when the health of a microservice goes down [45].

### 2.5.9 Resilience Value in Distributed Services

The key to resilience is the prevention of cascading failures. And the most effective patterns to combat cascading failures are Circuit Breakers and Timeouts. The cascading failures happen more often in microservices since the absence of resilience mechanisms can make one error jump into others and affect the complete ecosystem of interdependent microservices [16].

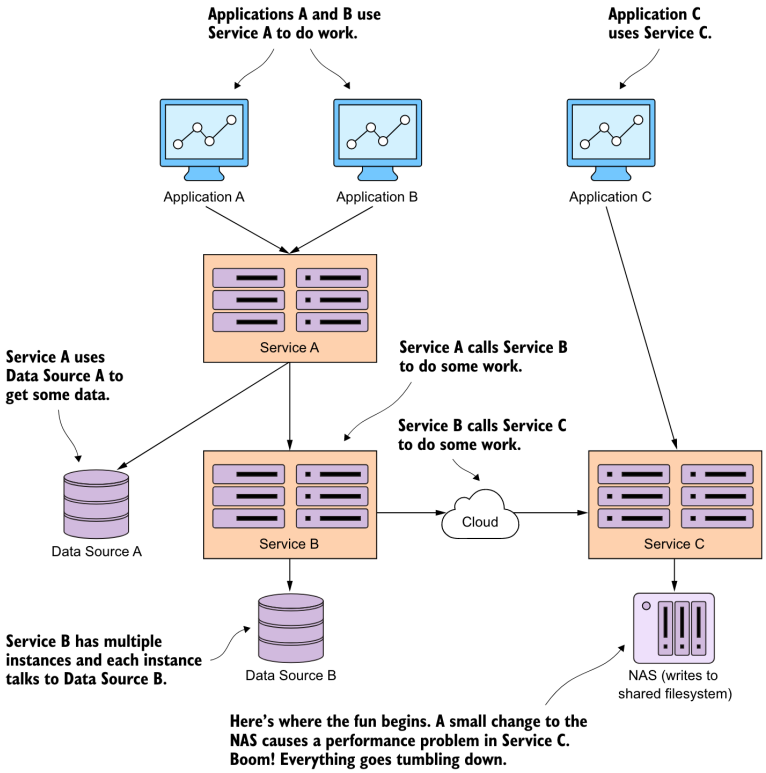



Figure 2.13 Transaction tracing diagram. Source: [15].

In the previous image, three different applications are using three different services. Microservice A uses its database, and it also uses microservice B. Microservice B uses its database and microservice C. Microservice C uses a NAS service.

In the absence of resilience mechanisms, a cascading failure starts with a change in the NAS service, which affects the microservice C and it jumps to microservice B, and microservice A, affecting all three applications.

The cascading failure could keep microservice B resources busy due to the absence of resilience mechanisms. Microservice B could keep open connections to its database while waiting for the slow microservice C. This happens in microservice B and will happen in the same manner in service A.

The scenario can be avoided with the correct resilience mechanisms. For example, a circuit breaker and a fallback value specifically in microservice C could prevent the cascading failure, by not waiting for the NAS service during its downtime.

#### 2.5.9.1 Testing Resilience in Distributed Services

Chaos Engineering is a way to test the resilience of a system. Just as we cannot guarantee that there will not be an outage in infrastructure, we cannot guarantee other critical issues unrelated to the application control, like sudden unexpected load, any type of latencies, and disruptions to known and unknown dependencies, will not happen. Understanding how the application is affected by these transient faults, is how we can manage and prepare the application for the unexpected. That is how resilience in an application is enhanced [46].

Such an uncommon approach to testing is required for microservices because even though different microservices in isolation can be safe, the composition, which is the nature of microservice architecture, is not necessarily safe [16].

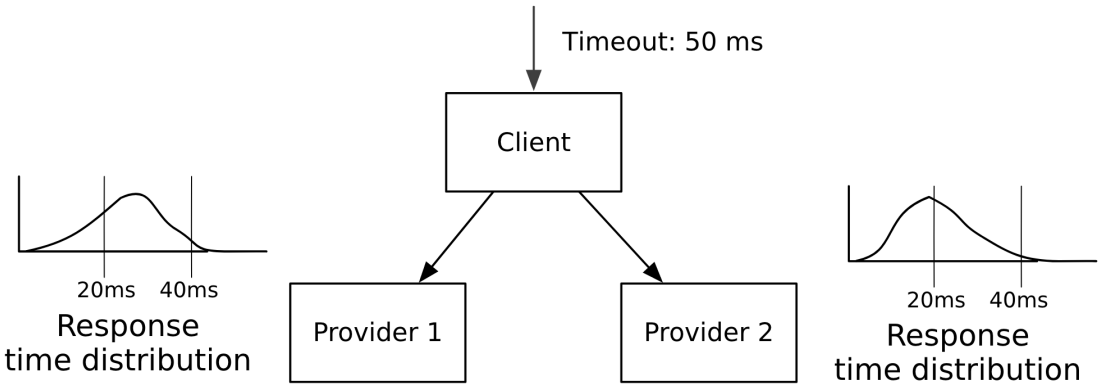



Figure 2.14 Isolation testing issue. Source: [16].

In the previous image, the client has a defined 50ms timeout. If we test each microservice in isolation, we can see the response time average does not exceed the timeout of the client. However, in a scenario where both microservices are used in sequence, a significant number of requests will exceed the defined client timeout.

## 2.6 Containerization

Containerization is the approach used to deploy software as images into virtual machines or physical computers. This approach brings many advantages, such as deploying the software and its dependencies, executed in its target operating system. Following this approach helps each application have its own [17].

The idea behind the containerization of software comes from the concept of the standard shipping container. Usually used to transport goods the container here transports software. It transports the software to a specific deployment, and allows more operations such as creation, start, stop, restart, and destruction [17].

This software that the containerization ships is represented by an image that developers can create very easily. Typically, developers use a base image, which consists of the required or preferred OS for the software. It is over this base image that the developer adds the software, including any library and dependency. The final image

is published, to be used later by the operations team, to create any number of containers and into the desired environment [17].

Containerization can be the first step to scalability since we can scale horizontally and create new instances of one service as required. The same is true for high availability and high resilience of a service. If one instance is crashed, or unavailable because of resources or any other different reasons, another instance can take care of the pending request [47].

The containerization also helps with the removal of a single point of failure for a single service. A single instance microservice running which crashes and is unable to recover but only by restart, leaves a gap in the application. Having more than one instance not only helps to fill that gap, while one of these instances is being restarted but also helps with the load balancing of the application [48].

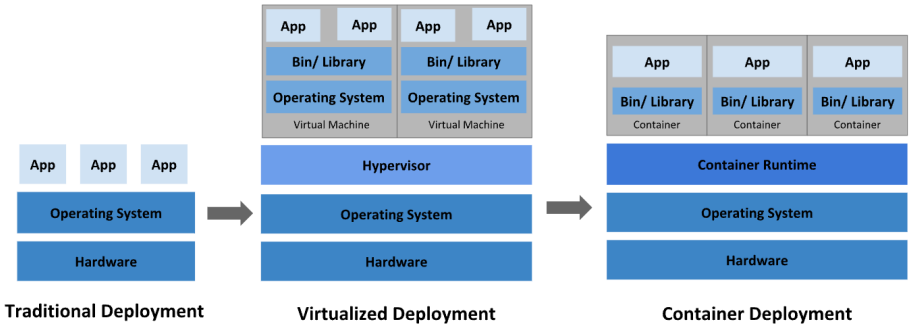



Figure 2.15 Containerization compared with traditional virtualization. Source: [49].

In the previous image, traditional deployment was the first way of deploying applications, directly into the operating system of the physical server, with no way to define resource boundaries between these applications. Resulting in resource allocation issues, and having some apps underperform due to others taking more resources.

With the virtualized deployment, resources could be allocated better. However with the container approach, we still have the virtualized deployment benefit, but we remove the hypervisor and operating system overhead, resulting in a better scalability regarding resource economy.

### 2.6.1 Docker

Docker, which uses containers, is an open-source tool that can automate the deployment of these. This tool gives you the ability to create images based on a software application, which can be published into public repositories and deployed into different environments [17].

Docker can help with the segregation of duties in the team. The developers' team can take care of the software that is running inside the container, while the

operations teams care about the infrastructure, which includes the health of each container, the high availability, and scalability [17].

Docker is also a great tool for the microservices architecture. The lightweight nature of containers makes it great for providing an ease of deploying a high number of applications [17].

### 2.6.2 Container Orchestration

With the need to manage, deploy, and debug many microservices, and multiple instances of each, the orchestration of containers is required. Container orchestration refers to the automated management, deployment, and coordination of the containers in different environments. With this tool, we can automate the replication for scalability per each microservice. This tool is also responsible for checking the health of each container restarting it if required and managing virtual resources, network, and persistent memory for each container [50].

#### 2.6.2.1 Kubernetes

Kubernetes is an open-source platform for managing container workloads and services [49]. It is a production-grade container orchestration tool, that offers various [51]. These features include load balancing with the use of local DNS to decouple containers communication between them, self-healing features that use a user-defined health check to restart any crashed container, horizontal scaling, and secret and configuration management [49].

Docker and Kubernetes have been found to have a great synergy together. They improve software development significantly by providing isolation between microservices, individual scalability, and efficiency. Leitner's study, shows this synergy, by increasing deployment frequency and reducing failure rates and recovery times against traditional virtualization strategies [52].

#### 2.6.2.2 Helm

Microservices architecture is an automation-heavy process that leverages container orchestration tools such as Kubernetes as presented above. It can become difficult to manage different applications with their very specific needs and configurations [53]. Helm is a tool for managing Kubernetes applications called charts, that represent the specific configuration, making it easier for developers, to upgrade and manage versions of these highly complex applications' infrastructure [54].

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Figure 2.16 Deployment declarative statement Kubernetes. Source: [55]

A declarative approach for the deployment of a web server is shown above. A complex application with distributed transactions could include more deployments, as a minimum of one per microservice. It also required other declarative statements, such as services that expose a network application that enables the link to dynamic ephemeral pods' dynamic IP addresses [18], per each microservice. Many other resources can also be needed as dynamic volumes, for stateful replication sets, making it harder to keep track of resources needed for an application when deploying, maintaining, or developing [18].

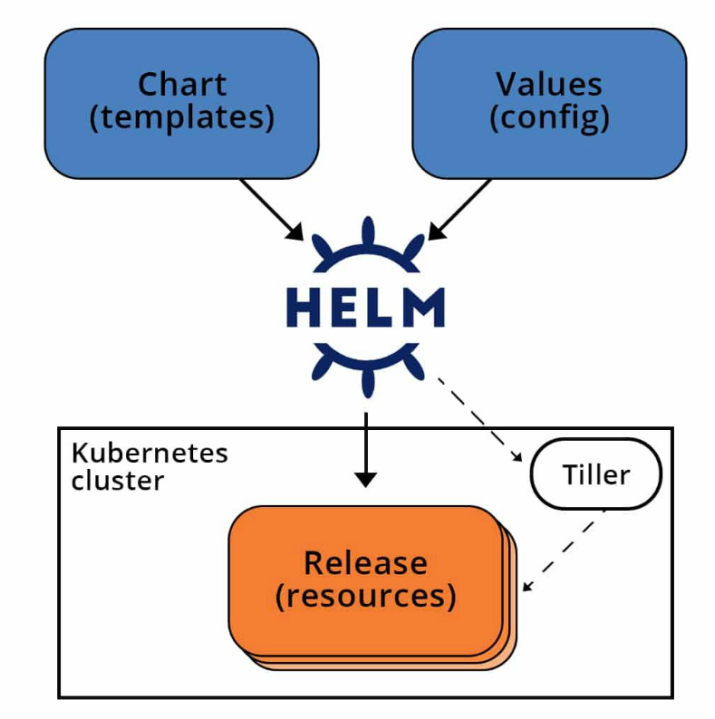


Figure 2.17 Rolling Update Kubernetes Diagram. Source: [56]

The way that Helm works with Kubernetes, is by declaring all resources needed by an application in one template. This template receives a set of values, used by the template that can be changed by deployment depending on the environment, infrastructure resources, or different criteria [57].

### 2.6.3 Kubernetes for Resilient Systems

Kubernetes' popularity as a container orchestration platform has risen and it's been used for managing applications in different domains, including cloud computing, edge computing, and specific cases of smart cities. These smart city cases have special requirements like high availability of real-time services. Examples are air quality, traffic, and weather monitoring systems. Kubernetes has been shown in the enhancement of not only real-time resource utilization but also tolerance [58].

Kubernetes has also been shown to enhance resilience in the cloud services. Azure Kubernetes Services, simplifies the management of complex applications, achieving high availability, scalability, and disaster recovery application setup. Kubernetes achieves high availability and scalability with the use of worker nodes of a cluster server, holding a defined number of replicas of the microservices and database services [59].

#### 2.6.3.1 Immutability

Kubernetes forces the developers to not mutate already running processes in any environment. The nature of Kubernetes that follows containerization, forces the replacement of the old container for a new one. Meaning that no changes would ever be required to be done directly (unless a very critical, necessary, and temporal fix is required) to a container [18]. This immutability ensures that a system is running consistently and as expected through time, making it easier to predict bringing reliability that contributes to resilience [30]. Although the focus of this research goes into managing unexpected failures, the prevention of these is also beneficial.

The immutability of infrastructure not only benefits resilience indirectly but forces any changes to be pushed through with new docker images. Which facilitates the rollback to a previous change, rather than undoing a complex set of imperative steps [18].

#### 2.6.3.2 Declarative approach

Imperative and declarative approaches are two different approaches in programming that define how computations should be executed. The imperative approach defines a sequence of steps required to achieve a desired outcome, while the declarative approach emphasizes the desired outcome and lets the underlying system achieve a defined approach. The declarative approach abstracts the process [60]. Because declarative just emphasizes the desired outcome, it can be understood without being executed resulting in a less error-prone process [18].

The declarative approach is tied to immutability, which enables the reproducibility of the same application, despite being in a different environment. This enables testing for resilience matters, knowing to expect the same behavior under certain tests in your testing and production environment. So declarative approach does not enhance resilience directly or indirectly but enables testing of an immutable system, which is required for testing to be valuable [18].

#### 2.6.3.3 Self-Healing

The previous declarative and immutable features of Kubernetes enable the self-healing feature. Self-healing features enable a system to autonomously detect failures, respond, and recover from disruptions without human intervention [61].

Traditional healing techniques involve monitoring, alarms, and human intervention in response to an event. This traditional healing technique involves a human non-immediate, error-prone part, an imperative process, that can lead to unexpected errors in each step [18].

Kubernetes' self-healing feature improves the development process because resources spent here will be spent in actual developing or debugging software-level errors. The reduced time in self-healing actions response time reduces the downtime and increases resilience due to the high availability of replicated microservices [18].

#### 2.6.3.4 Disaster recovery

Disaster recovery is a critical aspect of the resilience of an organization. This aspect encompasses the processes and strategies employed to ensure the continuity of operations in the aftermath of an unexpected disaster [62]. The importance of a disaster recovery plan comes due to the ability to improve the resilience of the system in the future [63].

Kubernetes provides an easy disaster recovery approach for the infrastructure with fault detection and recovery functions. These functions can be used together with an organizational disaster recovery plan, that will contribute to the building of resilient cloud infrastructure capable of withstanding and recovering from adverse events [18].

#### 2.6.3.5 Replicas

Kubernetes enhances resilience with its Horizontal Pod Auto-scaler (HPA) feature. This feature, monitors default resource metrics, such as CPU and memory usage of the nodes and pods [64]. When there is a decrease in load, the replicas are destroyed in a controlled manner to free resources for future use [65].

HPA enhances the resilience of the application, by reducing the probability of the request being failed, due to the unavailability of microservices instances. Microservices replicas make sure that there is at least a defined number of replicas always available to service incoming requests, making sure that every request is processed [65].

#### 2.6.3.6 Gateway API

The Kubernetes Gateway API is designed to manage any ingress traffic to the cluster. This API manages network traffic by using an extensible, role-oriented, protocol-aware configuration mechanism [66].

The Gateway defines three stable API Kinds, which are the following [66]:

1. Gateway: Defines an instance capable of handling traffic infrastructure.
2. GatewayClass: Defines a set of Gateways, with common configuration.
3. HTTPRoute: Defines a set of user-defined rules that handles HTTP traffic differently, based on certain criteria.


Figure 2.18 Resource model for Kubernetes Gateway API. Source: [66].

The flow in the previous image shows how Kubernetes Gateway API manages the traffic. In this example, the client sends an HTTP request and is managed by the Gateway, which is forwarded to an existing HTTPRoute, that contains HTTP-specific rules for mapping traffic.

The Gateway API helps with the enhancement of resilience by load balancing requests on the replica set of a microservice. Instead of an instance in a set of replicas getting all requests, the load is balanced through all instances of the replica set for better use of resources [66].

The Gateway can be also used as a resilience pattern. The circuit breaker pattern can be abstracted to the gateway API, to not direct traffic to degraded instances of a replica set. This pattern, with the equitable distribution of traffic of the load balancing feature, minimizes the risk of overloading specific microservice instances, enhancing the system’s ability to tolerate faults [66].

#### 2.6.3.7 Stateful Replicas

As we have seen, replicas enhance the availability and overall resilience of a service, against disruptions of services instances (containers). However, stateful containers, are different than stateless containers. Stateful containers (or containers that manage any type of state persistence) present challenges due to the container's on-disk files' ephemeral nature. Kubernetes introduces volumes, as a feature to manage on-disk files and the ability to share these between different container instances [67].

Database services require a persistent state, that is not tied to the container lifetime and is not tied to a container instance if a replica set would want to be used to enhance availability and resilience. Kubernetes introduces Persistent Volumes (PV) which are storage resources that live independently of the life cycle of an individual container [68].

Statefulset provides features that help with the database replication process as the following [69]:

1. Stable, Unique network identifiers.
2. Stable, Unique persistent storage.
3. Ordered, graceful deployment and scaling.
4. Ordered, automated rolling updates.
5.

The unique network identifiers are used to identify the place, order, or number of the container in the replica set, which even though is created from the same spec, is not interchangeable due to the data replication process [69]. It is also used for the DNS delegation name, for new instances able to encounter the previous container on the initial replication and further synchronization of the database [70].

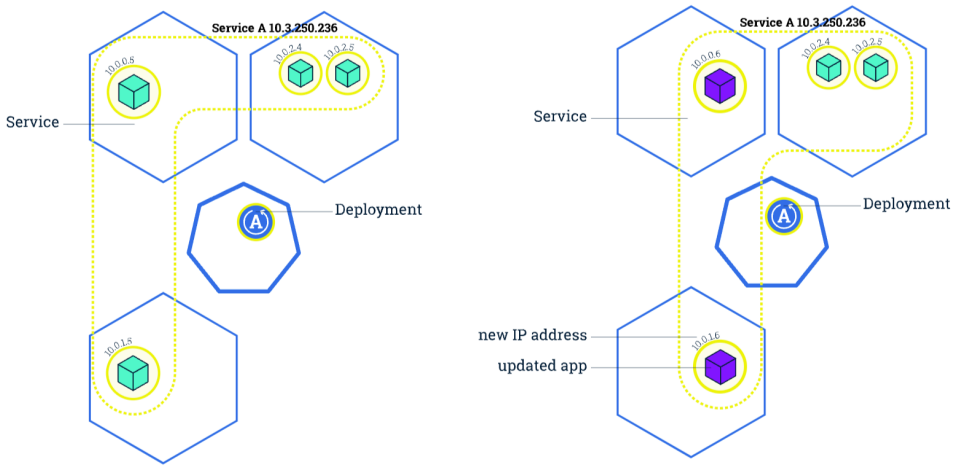



Figure 2.19 Rolling Update Kubernetes Diagram. Source: [71]

The previous image shows how a rolling update strategy works. Kubernetes compares the current state of a container against the desired. If the current state of that pod is not desired, it is updated, and the process continues with the next container until the update has been updated.

## 2.7 Database Replication

Database replication is a fundamental mechanism that ensures data availability, fault tolerance, and scalability in microservice architecture. This practice involves synchronization and replication of data between the new and already running containers [72]. As a result, database replication enables the ability of better scalability and fault tolerance by providing multiple copies of data in each container, ensuring that the service availability is high and fault tolerant during system disruptions [73].

### 2.7.1 MongoDB Replication

Mongo DB is a document-oriented NoSQL database that offers various features, including replication for redundancy and high availability with the use of one primary and two secondary members [74]. Not only has this feature been proven to be highly

functional, but it also allows the users to tailor the replication process to meet the needs of consistency while keeping a balance between the safety and performance requirements of the application [75].

Mongo DB replication architecture for NoSQL has been shown to have less impact on performance than other popular vendors like Cassandra. Cassandra replication has shown a degradation of 41.1% and 98% for read-heavy workload and write-heavy respectively, while Mongo DB has shown only a degradation of 33& and 52% [76].

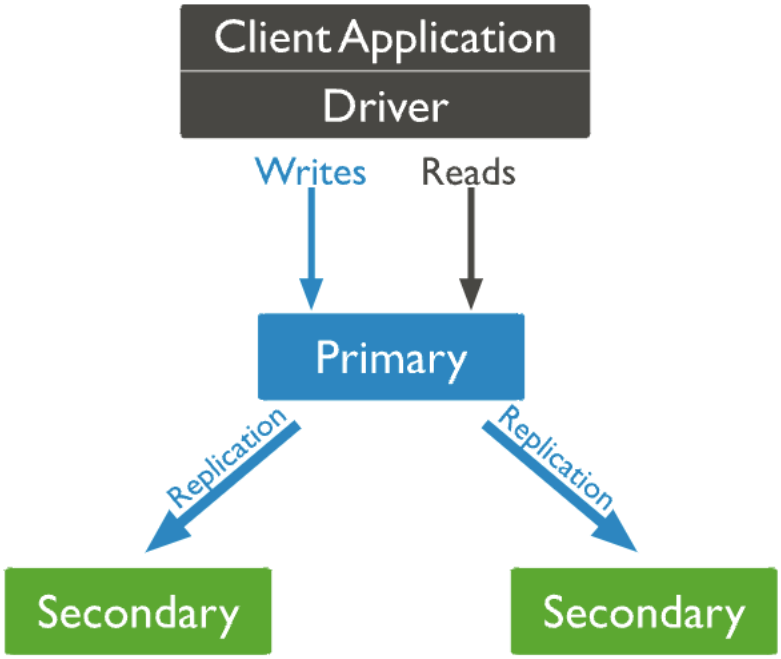



Figure 2.20 MongoDb replication model. Source: [77].

The image shows how the replication works between the three instances in the replica set. The primary instance receives all write operations and all secondaries replicate the data and operations log after each transaction, by default configuration [74].

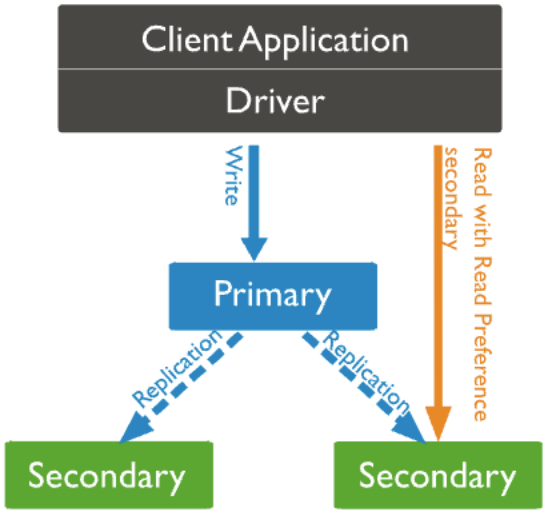



Figure 2. 21 MongoDb replication read operations. Source: [63].

The read operations are met by the primary instance; however, secondaries can also participate in redistributing the read load in the primary instance. This specific behavior can be set under the preferences configuration of the replica set [74].

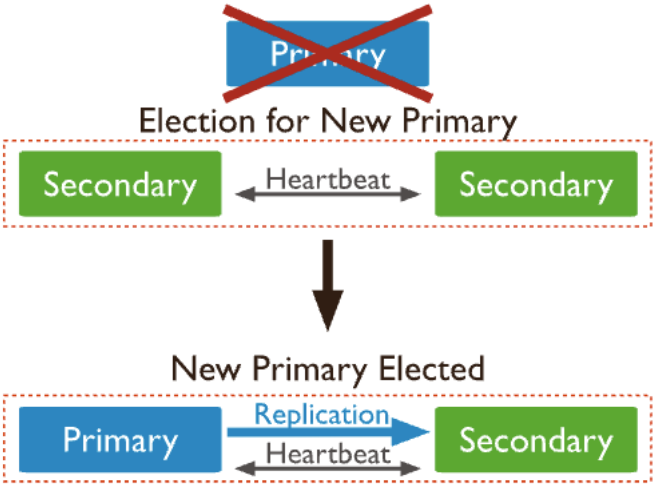



Figure 2.22 MongoDb replication automatic failover. Source: [63].

To detect the failover of the primary instance in the replica set, a heartbeat exists between each secondary and the primary. If a timeout of said heartbeat is detected, the automatic switchover of a secondary promotion to a primary is taken in action. The default timeout for a heartbeat is 10 seconds, but it can be configured by a replica set, which maximum time for an automatic failover action would be expected to be finished no more than 12 seconds. It is important to consider that a lower timeout would be beneficial for a faster switchover, but transient problems such as temporary network latency could force premature switchover of proper healthy primary instances, and these could result in rollbacks [74].

## 2.8 Chaos-Engineering

“Chaos Engineering is the discipline of experimenting on a system to build confidence in the system’s capability to withstand turbulent conditions in production.”[78].

With chaos engineering testing, we can check how an application behaves against unexpected events. The goal of chaos engineering testing is to enhance the application's ability to gracefully manage these events, by deliberately injecting these faults in a controlled manner through a series of experiments. In short, the main objective is to monitor the system's behavior and health, during fault injection experiments [46].

Many companies such as Amazon, Netflix, eBay, and Linkedln use microservice architectures to deploy, maintain, debug, and develop their large applications in the form of small, independently testable upgradable services. Nonetheless, these benefits come at a price of increased complexity for distributed transactions [79]. Even though complexity can be contained up until a certain point, the increase of it is imminent. That is why with Chaos Engineering discipline has gained popularity in bringing out to the light any vulnerability, preventing outages, and increasing the resilience of these complex distributed systems [21].

The history of Chaos Engineering starts at Netflix. Since cloud computing was still a new technology some AWS instances vanished abruptly, and the system needed to be ready to handle these outages. Chaos Monkey was one of the many tools that was tried, worked, and stuck around. This tool would pick up a random instance every day from a cluster and turn it off without any warning. The tool would do this during working hours, rather than late hours at night or early in the morning when human availability to fix these issues was low. [21]

### 2.8.1 Linear and Nonlinear Systems

Chaos engineering is a discipline for very specific systems. The system under test needs to be a complex system, distributed most of the time, that can benefit from controlled faults on different components to observe its behavior [21].

The way a system can be defined complex or not, can be attributed to the consequence of an action. A linear consequence means that an input dictates the magnitude of an output, making it easier to predict and test. While a nonlinear consequence, means that an input does not correlate with the magnitude of an output, making it more difficult to predict and test [21]. A distributed transaction that takes into account multiple microservices as well as third-party services, is considered a complex system since multiple outcomes can come disregard the input.

| Simple systems | Complex systems |
|---|---|
| Linear | Nonlinear |
| Predictable output | Unpredictable behavior |
| Comprehensible | Impossible to build a complete mental model |

Figure 2.23 Simple and Complex Systems. Source: [21].

The previous image shows, how the type of system is related to its output, and whether it is comprehensible or not within a mental model. It is easy to see why traditional testing methods for simple systems cannot be applied to complex ones since the behavior is unpredictable and does not concern the input. Chaos engineering testing should only be used in complex systems, for the sake of not overcomplicating testing for predictable simple systems [21].

### 2.8.2 Consistent Complexity

Even though Chaos Engineering should be used wisely to tackle complex applications, there is always some kind of complexity in all of these [21]. The first type of imminent complexity is accidental, or the one that we don't add because of a feature or added benefit. This accidental complexity is the byproduct of the compromise between time and code quality since we want to develop focusing on feature velocity, test coverage, and code health, but implicit priorities such as economics, workload, and time push against these. This compromise can lead to a suboptimal system with unrequired complexity. Even though complexity can be reduced at some point in time redirecting the workload of new features to refactoring, there is no reason to assume that the previous compromises were done in a less informed manner than the ones in the refactoring [21].

Essential complexity happens when new complex features such as high availability, flexible scalability, or distributed transactions are needed. This complexity is not accidental, but it is necessary to develop these new features [21].

### 2.8.3 Experimenting and Testing

Even though experiments and tests fall under the same umbrella of quality assurance in the context of software development, their objectives are different. Experiments are used to confirm hypotheses and the main objective aims to discover new knowledge. A test is a defined procedure to assess the quality or performance of a well-known system [80].

Integration testing would not bring the same results as chaos engineering experiments, since the tests need previous knowledge of what you are testing. Since we are not creating new knowledge, we are not discovering more about the unpredictable outcome of our complex system. Experiments, on the other hand, not only create new knowledge about a complex system but can also build confidence in an already existing system [21].

### 2.8.4 Verification versus Validation

Verification and validation are crucial processes in ensuring the accuracy of a system and the reliability of simulation models. The verification process is involved in the assessment of the correct implementation of the model of the system. On the other hand, validation is involved in the determination of the accuracy of the model of the system [81].

Because of the complexity of distributed systems, the Chaos Engineering discipline is focused on the verification of the system, rather than the validation. Even though the experiments bring over new knowledge about the system, the focus of verification over validation is because distributed systems could change how they interact with each other [21]. A good example could be any fallback pattern used for resilience enhancement that changes how microservices interact with each other to keep a system operational, meaning that validation would fail, but verification would succeed.

### 2.8.5 Chaos Engineering Principles

Chaos engineering principles are the following:

1. "Start by defining 'steady state' as some measurable output of a system that indicates normal behavior." [21].
2. "Hypothesize that this steady state will continue in both the control group and the experimental group."[21].
3. "Introduce variables that reflect real-world events like servers that crash, hard drives that malfunction, network connections that are severed, etc."[21].
4. "Try to disprove the hypothesis by looking for a difference in steady state between the control group and the experimental group."[21].

These principles can be extended to make them more efficient. The first principle can be extended to build a hypothesis around steady-state behavior. This means that the focus on the steady state of a system should be dictated by key performance indicators (KPIs). These indicators should reflect the output of the system, rather than the internal attributes of it. By focusing on these output indicators, the Chaos Engineering discipline verifies that a system works as expected, dictated by the steady behavior, rather than trying to validate how it works [21].

Vary Real-World Events is the next advanced principle that states that the variables in each experiment should reflect real-world events. Most of the time, experiments are run with easier variables rather than the ones that provide the most learning value, or variables are selected to reflect engineers' experience rather than users' experience [21].

Running experiments in production is another advanced principle that states that confidence is built in the environment you run it. This means that the confidence built by running in a staging environment does not translate fully to a production environment. Controversy arises when very delicate systems are tested in production,

but even then an incremental approach should be better than never running experiments in production since every environment presents different types of transient faults [21].

When you want to build confidence in a complex system, you need to keep running experiments continuously since the system is affected in the same manner, by maintenance or development. That is why automating experiments to run continuously is another advanced principle to keep this discipline bringing value to the table [21].

Finally, minimizing the blast radius is the last advanced principle that states that a good Chaos engineer can ensure the fallout from experiments is minimized and contained [21].

### 2.8.6 Chaos Mesh

Chaos Mesh is an open-source cloud-native Chaos engineering platform. This platform offers all the required tools to be used in synergy with Kubernetes to set up experiments that can uncover failures in the selected environment as development, testing, or production. Chaos mesh has many strengths that make it a good selection for this type of experiment. It was developed using the core library for testing TiDB, a distributed redundant open-source SQL service [82]. It provides a web UI, to create, manage and monitor your chaos engineering experiments. Finally is widely used in numerous companies and organizations like Tencent, Meituan, Apache, APISIX, and RabbitMQ [83].

#### 2.8.6.1 Chaos Mesh Architecture

Chaos mesh uses Kubernetes Custom Resource Definition, extension resources to the Kubernetes API [84], as a base to be included in Kubernetes. Chaos mesh has three principal components which are the following [83]:

1. Chaos Dashboard: is the user-friendly UI component that enables the management and monitoring of the experiments.
2. Chaos Controller Manager: This is the core logical component of Chaos mesh, that uses the Kubernetes Controller feature to schedule and manage the experiments.
3. Chaos Daemon: This is the core execution component of Chaos mesh, which uses the Kubernetes DameonSet feature that runs in a privileged mode to interfere with specific network devices, kernels, file systems, and other features in a pod.

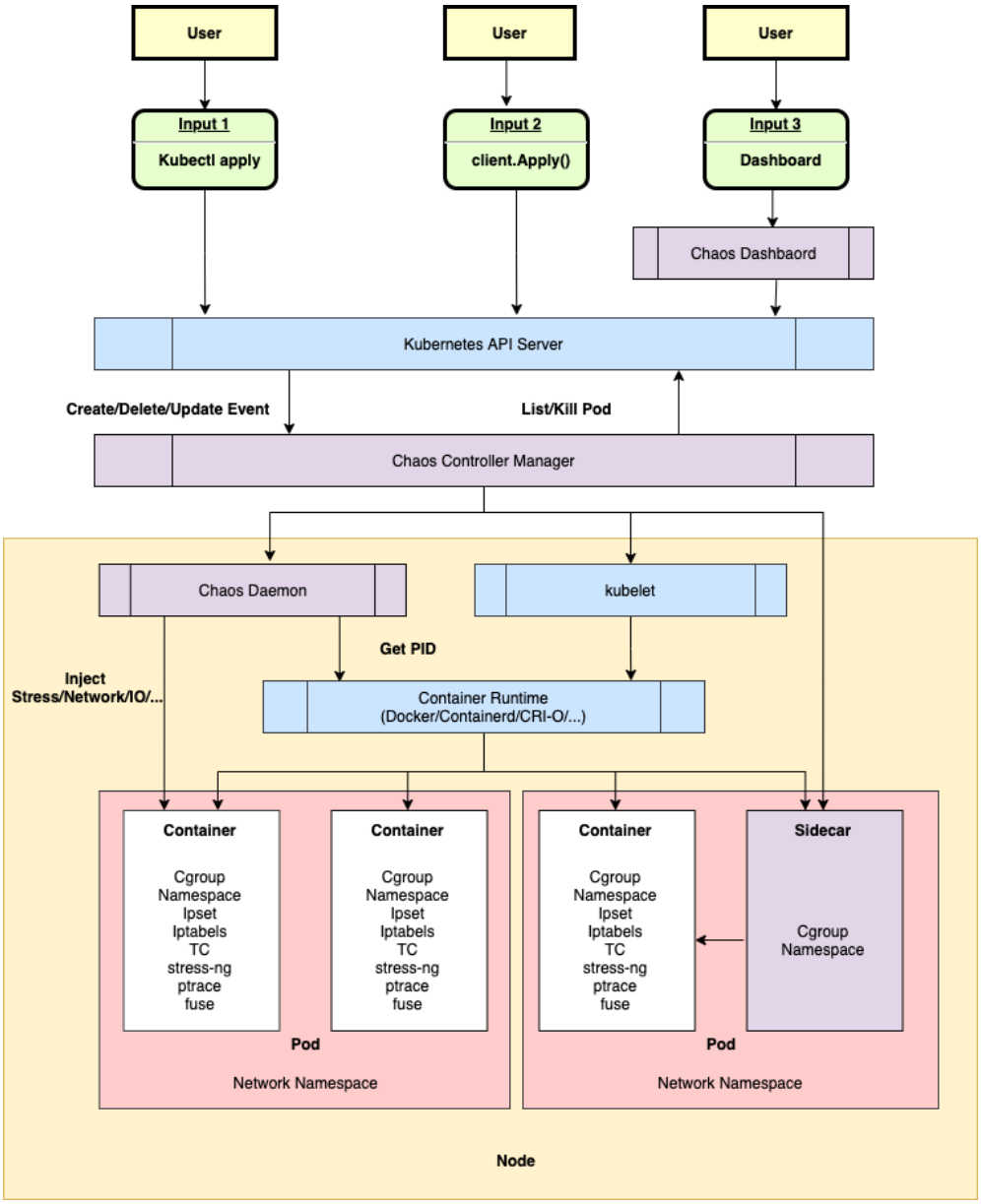



Figure 2.24 Chaos Mesh architecture overview: [83].

In the previous image, the overall architecture of Chaos mesh can be seen. Input of users can be used by Kubernetes API, or Chaos Dashboard, but never directly into Chaos Controller manager. The Chaos controller manager manages the Chaos daemon that can do different types of privileged actions such as stress-ng, iptables, and different to simulate real-life failures.

#### 2.8.6.2 Pod Faults

Chaos mesh platform enables pod faults with PodChaos to simulate real-life disruptions due to pod failures such as a Pod node restart, a Pod's persistent inability, and specific failures per container in a Pod. When doing experiments with this type of failure, some previous notes need to be considered such as, not having a Control Manager of Chaos mesh in the target Pod, and if killing a Pod, recovery mechanisms are configured to ensure that the pod availability can be restored [85].

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: PodChaos
metadata:
  name: pod-failure-example
  namespace: chaos-mesh
spec:
  action: pod-failure
  mode: one
  duration: '30s'
  selector:
    labelSelectors:
      'app.kubernetes.io/component': 'tikv'
```

Figure 2.25 Chaos Mesh Pod-Failure example: [85].

This example shows a Pod failure definition, that targets a pod with the component label ‘tikv’, lasting for 30 seconds. This failure can be used to simulate a complete service outage, such as a third-party service or a complete microservice.

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: PodChaos
metadata:
  name: pod-kill-example
  namespace: chaos-mesh
spec:
  action: pod-kill
  mode: one
  selector:
    namespaces:
      - tidb-cluster-demo
    labelSelectors:
      'app.kubernetes.io/component': 'tikv'
```

Figure 2.26 Chaos Mesh Pod-Kill example. Source: [85]

This example shows a Pod Kill definition, that destroys a targeting pod. Duration is not defined here, because the failure type is different from the previous, where here the Pod is destroyed for recovery mechanism as self-healing Kubernetes or horizontal scaling can be tested.

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: PodChaos
metadata:
  name: container-kill-example
  namespace: chaos-mesh
spec:
  action: container-kill
  mode: one
  containerNames: ['prometheus']
  selector:
    labelSelectors:
      'app.kubernetes.io/component': 'monitor'
```

Figure 2.27 Chaos Mesh Pod-Container-Kill example. Source: [77]

This example shows a Pod Container Kill definition, that kills a specific container in a Pod. This failure can be used to target a specific container in a Pod such as in a high-availability database cluster, to target the master node and test the takeover from a slave node.

#### 2.8.6.3 Network Faults

Chaos mesh platform enables pod faults with PodChaos to simulate real-life network transient failures. These failures include partitions, that is the division between applications in the network. Net emulation is the simulation of different network conditions, such as high delay, high packet loss rate, packet duplicating, packet corruption, and others. Finally, there is the Bandwidth failure type that emulates the bandwidth limitations you could encounter in real life [86].

Before executing these network failures, we must consider that the Chaos Controller and Chaos daemon connection need to be always operational since these failure types are not timed by nature, so if the connection is lost, there is no way to restore the normal state of the system [86].

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: NetworkChaos
metadata:
  name: delay
spec:
  action: delay
  mode: one
  selector:
    namespaces:
      - default
    labelSelectors:
      'app': 'web-show'
  delay:
    latency: '10ms'
    correlation: '100'
    jitter: '0ms'
```

Figure 2.28 Chaos Mesh Network emulation example. Source: [86]

This image shows the network emulation declaration. This is not a disruption of the network, but a modification of its normal state, hence it is called emulation and not failure. This declaration targets the pod with the app label of ‘web-show’ and modifies the network to have a latency of 10 milliseconds. Other types of actions can also be introduced such as packet duplication, packet corruption, network partition, and network bandwidth limit [86].

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: NetworkChaos
metadata:
  name: partition
spec:
  action: partition
  mode: all
  selector:
    namespaces:
      - default
    labelSelectors:
      'app': 'app1'
  direction: to
  target:
    mode: all
    selector:
      namespaces:
        - default
      labelSelectors:
        'app': 'app2'
```

Figure 2.29 Chaos Mesh Network Partition example. Source: [78]

This image shows the Network partition declaration which blocks the network traffic from App1 to App2. The direction of the traffic can also be defined to test specific scenarios, such as a new firewall policy applied without proper notice.

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: NetworkChaos
metadata:
  name: bandwidth
spec:
  action: bandwidth
  mode: all
  selector:
    namespaces:
      - default
    labelSelectors:
      'app': 'app1'
  bandwidth:
    rate: '1mbps'
    limit: 20971520
    buffer: 10000
```

Figure 2.30 Chaos Mesh Network Partition example. Source: [78]

This image shows the Bandwidth limit declaration, which simulates bandwidth limitation that normally occurs daily due to poor network management or poor quality of service. Here the declaration targets the pod with the label ‘app1’ and limits its bandwidth to 1 mbps.

#### 2.8.6.4 Stress Scenarios

Chaos mesh platform enables stress scenarios simulation with StressChaos, to simulate real-life scenarios of high CPU or memory usage of the containers. This is one of the most valuable tools to test for scalability and load management [87].

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: StressChaos
metadata:
  name: memory-stress-example
  namespace: chaos-mesh
spec:
  mode: one
  selector:
    labelSelectors:
      'app': 'app1'
  stressors:
    memory:
      workers: 4
      size: '256MB'
```

Figure 2.31 Chaos Mesh Stress simulation example. Source: [87]

This image shows the declaration of a stress simulation that could lead to failure. The example targets a pod with the app label of ‘app1’ and applies 4 threads that take the 256 MB of memory. Depending on the amount of memory the containers have, this could lead to degradation and even a complete unavailable of the service.

#### 2.8.6.5 AWS Faults

Chaos mesh platform enables AWS faults simulations AWSChaos such as EC2 instances failures as stop, restart and volume detach from EC2 instances. AWS has become one of the leading web platforms that holds up to 40% of the cloud-server market, which is why AWSChaos aims to test one of the most used cloud infrastructure platforms [88].

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: AWSChaos
metadata:
  name: ec2-stop-example
  namespace: chaos-mesh
spec:
  action: ec2-stop
  secretName: 'cloud-key-secret'
  awsRegion: 'us-east-2'
  ec2Instance: 'your-ec2-instance-id'
  duration: '5m'
```

Figure 2.32 Chaos Mesh AWS EC2 failure simulation example. Source: [79]

This image shows the declaration of an AWS fault, which targets a cloud virtual computer instance in a specific region for a specific time. This type of fault can be beneficial to test since cloud providers can promise a high availability, but they are always susceptible to disasters that could affect specific regions [89].

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: AWSChaos
metadata:
  name: ec2-detach-volume-example
  namespace: chaos-mesh
spec:
  action: ec2-stop
  secretName: 'cloud-key-secret'
  awsRegion: 'us-east-2'
  ec2Instance: 'your-ec2-instance-id'
  volumeID: 'your-volume-id'
  deviceName: '/dev/sdf'
  duration: '5m'
```

Figure 2.33 Chaos Mesh AWS volume failure simulation example. Source: [79]

This image shows the declaration of a volume detach failure of a virtual computer. Since the container's state is ephemeral, database services need persistent data which brings the need for volumes. The AWS volume detach simulation can be used to test data backup strategies as the fair load and data distribution to ensure fault-tolerant data backup [90].

## 2.9 Message Brokers

Microservices need to be independent and able to communicate with each other easily, without tying to a specific protocol. Communication of many microservices can become messy and unreliable if the incorrect protocol is selected. HTTP protocol, a common communication protocol, even though, used widely can present certain problems with long-running tasks, since HTTP timeout has a timeout. Message brokers play a crucial role by enabling the communication of these microservices in a decoupled and asynchronous [91].

Moreover, another main benefit of resilience enhancement of message brokers is the ability to store messages where there is no short living time and where different instances of a microservice can serve different requests [91].

### 2.9.1 RabbitMQ

RabbitMQ is an open-source message broker that acts as an intermediary between the different microservices. RabbitMQ allows the reduction and distribution of load by delegating to different services, where they would typically take longer [91]. Furthermore, RabbitMQ provides a better performance against REST Apis with many parallel requests, in the context of a microservice architecture. Showing that message brokers for a microservice architecture is beneficial. [92]

RabbitMQ has many benefits which are:

1. Open Source: it was originally written in Erlang language, and it was released under the Mozilla Public License. Now owned by Pivotal Software Inc., RabbitMQ can benefit the open-source community for enhancements and add-ons, while Pivotal provides the commercial benefits of commercial support for ongoing product maturation [19].

2. Lightweight: RabbitMQ only uses 40 MB of RAM to run, plus any additional messaging queue that will take additional space [19].
3. Plugins for higher-latency environments: RabbitMQ addons enable communication with different environments. Since local environments have low latency, but services can be hosted in external networks, the plugins enable RabbitMQ to be clustered in the same network and share federated messages across data centers [19].
4. Client libraries for most modern languages: This is very important since it does not limit the ability to work with different technologies and languages in a microservice architecture. RabbitMQ's support of multiple languages like Java, Python, PHP, JavaScript, and C#, is the reason for it being used as the centerpiece between applications in the microservice architecture [19].

Many different companies use RabbitMQ today. These companies include Reddit, whose core platform relies on RabbitMQ to serve millions of web pages per month. Any event such as registering, new posts, upvotes, and any others is published into RabbitMQ’s queues to be processed in an asynchronous matter, enhancing performance and user experience [19].

Other very important entities use RabbitMQ. NASA chose RabbitMQ to be the message broker for their Nebula platform. This platform, used for the central infrastructure, later grew into the OpenStack platform, used for private and public cloud services [19]. Also, India’s biometric system relies on the RabbmitMQ message queue to relay messages to its different tools, such as monitoring, data warehouses, and more, to provide a system that serves 1.2 million people [19].

#### 2.9.1.1 Advanced Messaging Queueing Model

RabbitMQ is based on the AMQP model (Advanced Message Queuing Protocol), which is an open internet protocol used in business messaging between applications and organizations. AMQP plays a crucial role in communication in modern computer systems, distributed systems, and cloud computing systems [93].

RabbitMQ takes advantage of using these protocol principles to build the correct messaging system for each system. RabbitMQ can route messages to different queues through exchanges that define dynamic routing [19].

Exchanges are the first component in the AMQP model. This component is responsible for receiving messages and routing them by custom criteria. These criteria mostly are data attributes and message properties [19].

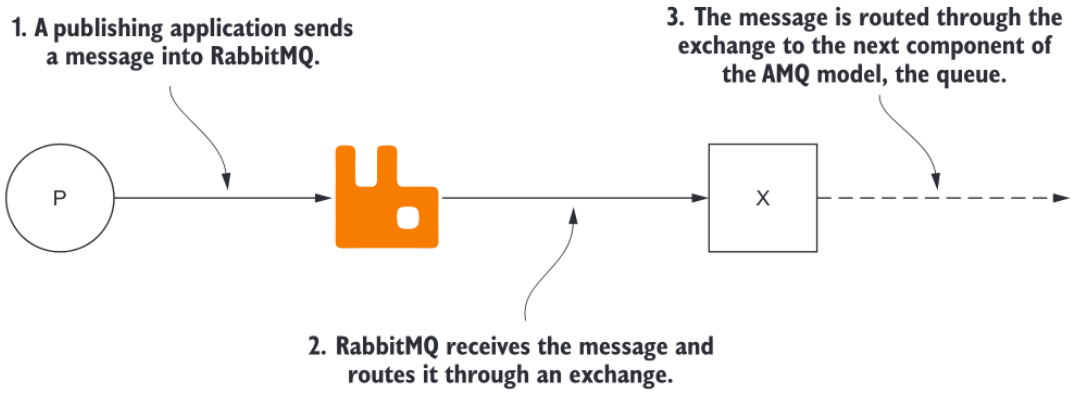



Figure 2.34 Exchange element in RabbitMQ. Source: [19]

The image shows the three steps of the most basic interaction of RabbitMQ. In this interaction, the exchange is responsible for the second step, by routing the message to a specific queue depending on some defined criteria. Exchange is one of the most powerful components since it can help solve very specific problems such as conditional messaging, and subscriber with publisher patterns [19].

Following this component, we can find queues. Queues are the component where messages are stored for future consumption. Queues can store messages only in RAM for fast management, or they can persist in disk memory with compromised performance. Queues work in a FIFO (first in, first out) strategy [19].

RabbitMQ queues are used to avoid a resource-intensive task immediately and to not overload the system. These queues are used to schedule tasks, and when many instances of a service are running, these tasks can be shared by each instance, doing load balancing. The queue is limited by the memory and disk space of the host. Moreover, work queues are used mostly when requests take longer than an HTTP request timeout window [94].

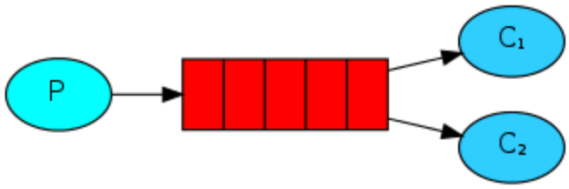



Figure 2.35 RabbitMQ Work Queue. Source: [94]

The image shows the architecture of a work queue, where a service named ‘P’ produces requests and two instances from the same service type called ‘C’, consume the requests. You can see how this enables better scalability in the long run since another instance can be added automatically enabling the complete system to handle a bigger load.

Finally, binding is the last component in the RabbitMQ messaging model. Bindings are responsible for defining in what queue should a message be sent. This is

where multiple bindings can exist, or specific to provide business logic that benefits the system [19].

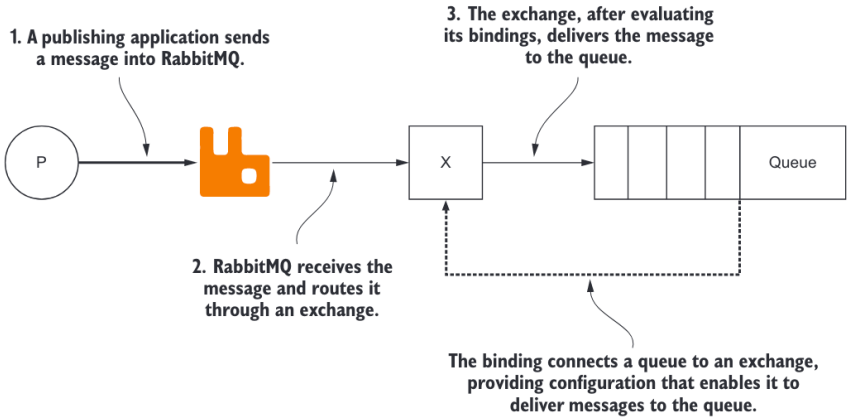



Figure 2.36 Exchange element in RabbitMQ. Source: [85]

The image shows how a binding defines how the exchange decides to what queue should the message be sent [85].

#### 2.9.1.2 Decoupling systems

Decoupling software systems can benefit from enhanced resilience while working with complex systems [10]. The use of diversity and redundancy of microservices instances enhances resilience but can increase coupling in systems. RabbitMQ can be used as a message-oriented middleware to reduce coupling [19].

"Message-oriented middleware (MOM) is defined as software or hardware infrastructure that allows for the sending and receiving of messages from distributed systems. RabbitMQ fills this role handily with functionality that provides advanced routing and message distribution, even with wide area network (WAN) tolerances to support reliable, distributed systems that interconnect with other systems easily." [19].

Tightly coupled architecture can cause a service to halt, by waiting into another. These in a complex orchestrated architecture such as Saga Pattern can affect the subsequent steps. However, with a message-oriented middleware, a tight-coupled application can be easily decoupled. This means that now services do not need to wait for database writes, third-party services, or long tasks to finish, but they can now just wait, freeing resources for other requests [19].

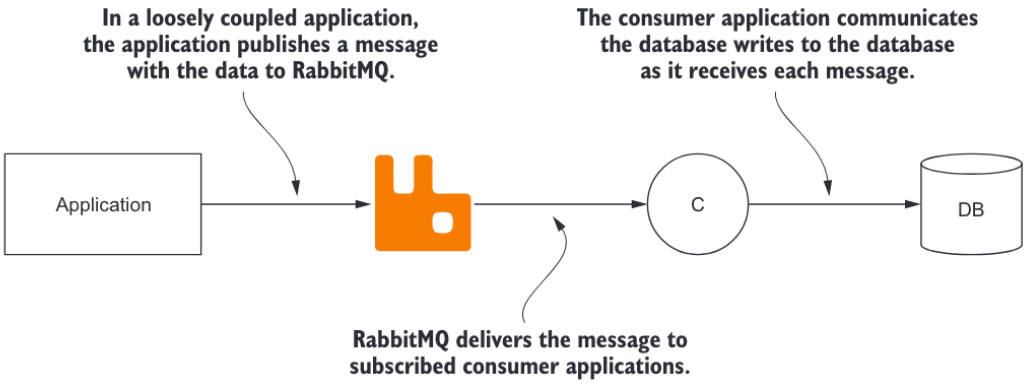



Figure 2. 37 Loosely coupled architecture with RabbitMQ. Source: [85]

The image shows an example of decoupling database writes. Instead of having the application do database writing directly and having the opportunity to keep resources busy while this happens, the database writing is delegated to a different service of message-oriented middleware. The new service takes responsibility for what the application previously had, enabling the main application to have more availability and performance [19].

## 2.10 System Under Test

The system under test designed for this study encloses a distributed system transaction that implements a microservice architecture. The distributed transaction to study requires multiple steps that commit to the database and third-party services on each step. A very well-known process is a simple e-commerce purchase order. In this transaction, many steps need to be done to process a purchase order, such as checking the inventory availability, processing the payment, keeping a record of the payments, creating shipping orders, and notifying the customer that their order was processed successfully.

The e-commerce example system that is under test is designed with the idea of third-party services involvement to best simulate the real-world scenarios of software architecture. These third-party services can include payment services, notification services, or any other that is not under the control of the software architect. Also, steps required for the process of the order are segregated into its domain to create microservices, each responsible for its proper entities and business domain logic.

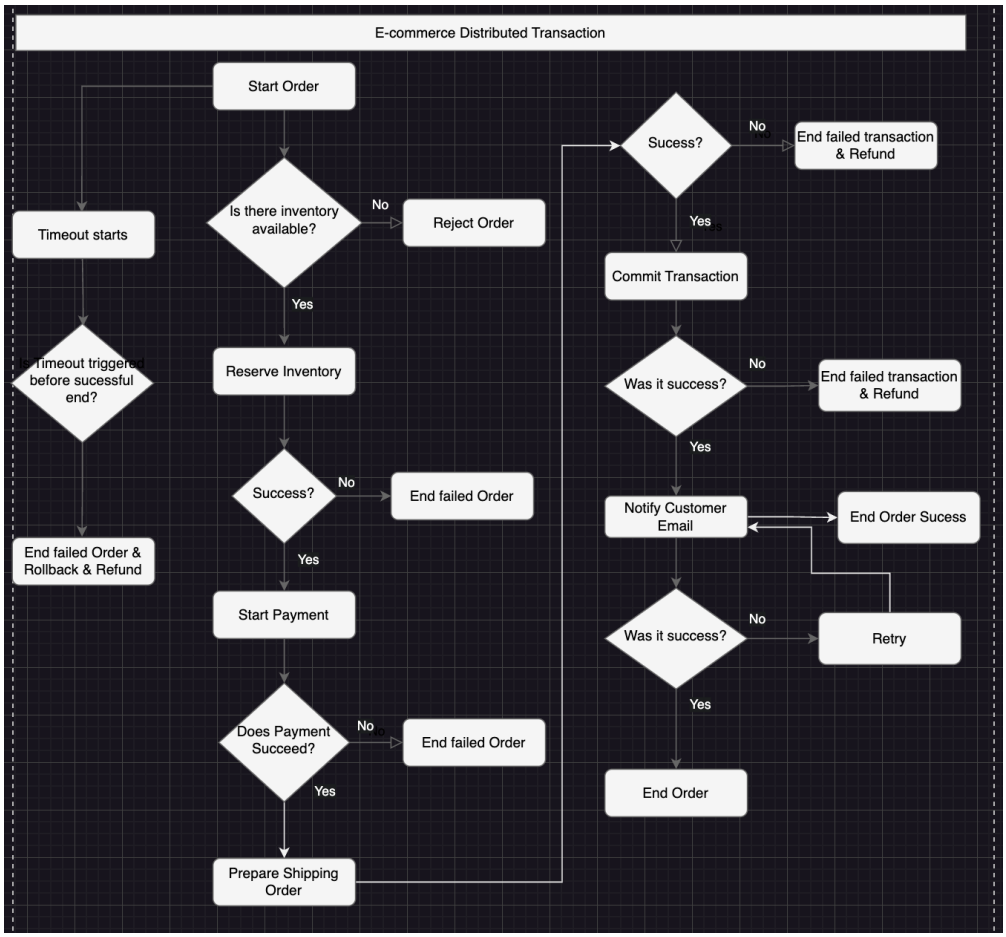



Figure 2.38 Workflow for E-commerce distributed transaction. Source: own

The previous image shows the workflow of the distributed transaction used for the design of the development of the system under test. The workflow defines the steps to be taken to successfully place a purchase order in an e-commerce system. It also defines the outcomes of each possibility.

To explain the workflow of registering an order placement, we can foresee the requirements of an order. A purchase order requires a payment transaction, a check of inventory, a shipping order depending on the business model, and a notification to give feedback to the client for a long-running process.

The workflow starts with a check of inventory, that when successful, continues with a reservation of inventory. This means that the process reserves this item for the current instance of the workflow. In case the process fails at any point after the reservation of items, the system should make the previously reserved items available again. The process of payments starts when the inventory is reserved and if successful, the shipping orders are created as a last crucial step. After this last crucial step of creating shipping orders, comes a non-crucial step, which is the notification. This noncrucial notification step is left at the end because the failure of these can be managed as a retry strategy or a later schedule. Nonetheless, notification failure is not a crucial step, meaning that the order can be processed successfully even with a notification failure.

All these steps, need to happen in an orderly manner. Meaning that this is not a parallel group of tasks, but rather they need to be completed serially. Moreover, any specific task can fail, due to transient faults and it needs to be gracefully handled. That is why there are many “End failed Orders”, which means that a crucial step has failed, and the process cannot be continually continued. It would be not logical to start a payment transaction if the item to purchase does not have enough quantity or is no longer available. However, this does not mean that all steps are crucial. Notifications, even though important, can be easily affected by communication issues outside the system's scope. For example, an email being marked as spam, or a customer not having an internet connection to receive a notification. These noncrucial steps of the process can be forsaken, for a later reply or left at the end for the sake of having a better chance of a successful request process.

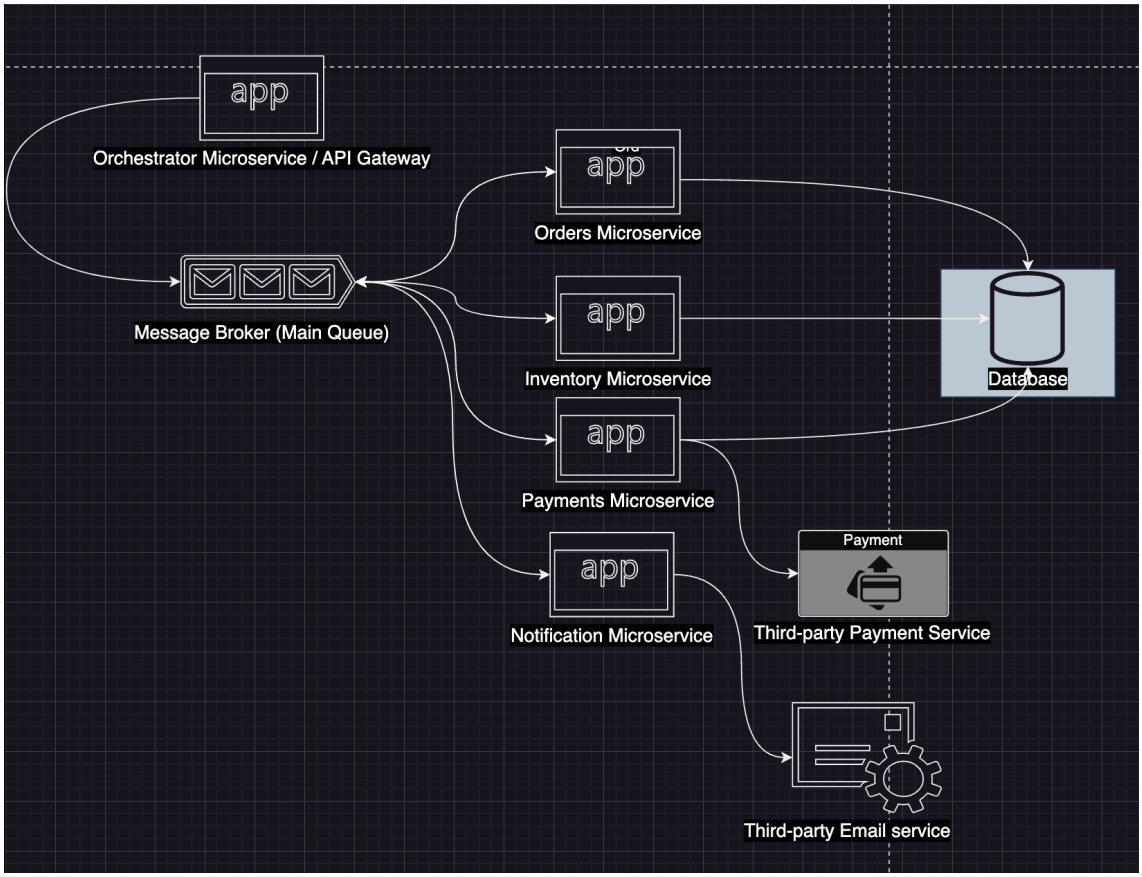



Figure 2.39 Architecture for e-commerce distributed transaction. Source: own

The image shows how the domain of each part related to the transaction of the process of a purchase order is divided into different parts. Each is represented by a microservice that will work together with the others to complete the distributed transaction. The architecture shows an API gateway and orchestrator service together since the load of an orchestrator and API should grow simultaneously at an equal rate. The same cannot be said for the other services, which can be used multiple times for the processing of a request, depending on the result of a request.

Moreover, the architecture shows the use of a message broker, to decouple each microservice and not depend on more HTTP request windows, other than the first open

between the API gateway and whoever is requesting the purchase order. We see two third-party services included in the overall architecture. These represent services that can present failure or timeouts that cannot be managed but should be expected. Finally, there is a Database to persist data related to the purchase order.

### 2.10.1 NserviceBus

NserviceBus is an extensible framework, that leverages complex principles of the service-oriented architecture to create reliable, extensible, and scalable distributed systems. NserviceBus makes multiple distributed transactions easier, by utilizing principles of consistency to reduce normal issues in distributed transactions such as database locking and blocking. This framework solves the aforementioned problems or microservice architecture, allowing the architect and developers to focus on business concerns [20].

Long-running processes are common in business. They tend to always exist in any type of business requirements, due to big waiting times, but also in waiting for other events that can happen anytime in the future. Normally these tasks are dealt with as batch jobs [20].

Batch jobs are usually executed at low load hours, usually at night when users do not use the system. However, these batches are prone to errors and have other limitations such as a limited amount of time which become shorter each time, due to the growth of the workload. Even other businesses have no off hours, meaning that running batch jobs will always impact the system performance regarding the time they are executed [20] Even when doing batch jobs, users would need to wait for the next day to see expected changes in the system.

Sagas, as discussed previously, are orchestration of processes that are not limited to the size of an HTTP request timeout window. Long-running processes can be executed by sagas, and with NServiceBus each saga has its saga data, that represents the status of a long-running process [20].

Since a Saga is a long-running process, it requires knowing what step of the process it is, otherwise, the system would be required to run each instance of the process from beginning to end, where not all long-running processes have a defined end. NServiceBus provides a configurable persistence layer, where it stores data related to sagas and timeouts [20].

//todo move after discussing everything about services
NServiceBus is used for the development of the system under test since it provides a complete framework for designing and testing sagas with native resilience patterns as previously discussed.

#### 2.10.1.1 Error queue and retry strategy.

Sagas, as previously defined as equal to long-running tasks, can present failures at each step, which means that the longer the task, the more probable it is for it to present a failure. In the case of the e-commerce workflow, a third-party service such as the payment can fail, forcing the complete saga to roll back. However as previously

discussed, we know that transient errors are common in real life, and before giving service as an unavailable, and more importantly one that is third-party, we can retry and hope for a successful resolution, instead of having to roll back all the previous steps taken.

NServiceBus provides a retry resilience pattern. To recover from these transient, or systematic exceptions, an error queue is defined by NServiceBus, to retry immediately or after a defined time. NServiceBus can retry this, in two different manners, depending on the type of failure. NServiceBus can either retry immediately when dealing with transient errors since these are not common to be long-lived errors. On the other hand, after the immediate retry of requests that fail the same way, it is concluded it is not a transient error, and NServiceBus delays each retry [95].

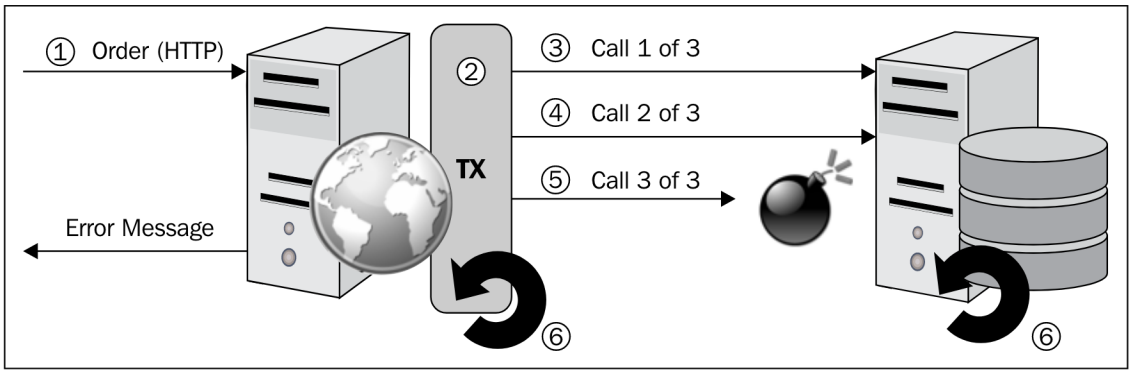



Figure 2.40 Saga transient failure without retry pattern. Source: [20]

This image shows how a saga would run if a step would fail. The third step here fails, forcing the rollback of the previous two steps to the database. Even though data consistency is not compromised, the failure of a transient error in step three can be the reason why this purchase order is not completed, and not able to follow ahead.

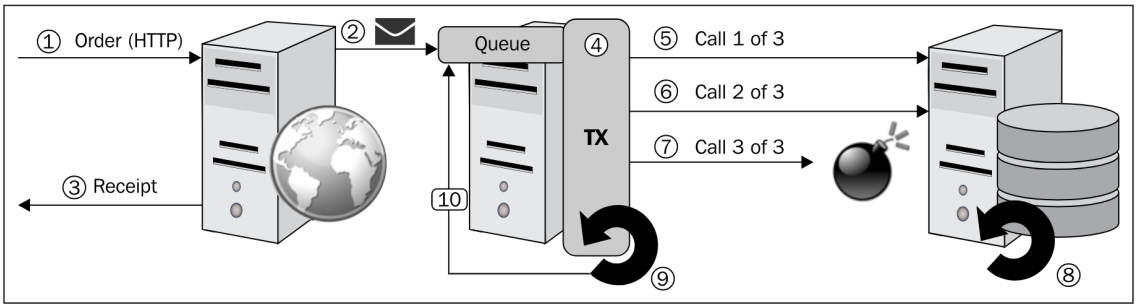



Figure 2.41 Saga transient failure with retry pattern. Source: [89]

This image shows the same previous scenario, except that now it includes the native retry strategy of NServiceBus. Here even though a failure happened in step three, the saga orchestrator can retry this step, instead of doing a rollback of the previous two steps. Should the failure be of transient type, or a short-lived failure, the saga can continue, and the purchase order could be processed successfully.

#### 2.10.1.2 Persistence for Single point of failure

As seen, Saga orchestration represents a single point of failure regarding the orchestration of messaging. This means that the state of the saga can be lost if the orchestrator is destroyed by the containerization orchestrator. That is why NServiceBus uses persistence, not only for data storage of saga states and other features but also to remove the possibility of losing a saga process [96].


Figure 2.42 Saga data state with NServiceBus persistence. Source: own

The image shows an instance of a saga. The instance, represented by a MongoDB document, shows the state of the process. Each saga has its data, but this example shows flags that represent if rollback has been applied, to different steps of the saga. This data persists, even if the orchestrator and other services are destroyed, or restarted, the orchestrator will be able to continue with the already living sagas [20].

#### 2.10.1.3 Timeout

NServiceBus also offers native functionality for timeouts, which is a resilience pattern as previously discussed. Even though sagas are long-running processes, should they exceed a defined time window, NServiceBus can trigger a timeout. This timeout means that the process is being executed, and it could either be left as it is, or a rollback of committed changes can take place [97].

### 2.10.2 NET

.NET is a free, open-source, cross-platform to develop different types of applications. Its library repository, has over 100,000 packages, making it easier to develop any specific application required by the business. Since it is open-source, .NET has an extensive community to help with problems [98]. .NET has been so widely accepted that it has been used by big companies such as UPS, Stack Overflow, NBC, and others [99].

.NET is also, the better option compared to other robust frameworks such as Java Platform EE, where .NET came up with better communication request handling

giving on average two times higher. These advantages can be attributed to a high level of performance optimization in the .NET framework libraries [100].

.NET is used to develop the system under test since it not has only shown its better performance for microservices architecture, but also it presents other benefits such as cross platforms, that can be used for containerization in Linux or Windows containers [101]. NServiceBus also works in synergy with .NET, using its platform to enhance microservice architecture development [20].

## 2.11 Metrics

To determine if a system has been improved or not, the reliance on metrics has been widely adopted. However, these metrics need to be considered by their effectiveness depending on the strong correlation to what aspect of the system is improved [102]. For resilience matters, metrics that provide observability in the performance of a system can be taken as strong correlation metrics [13].

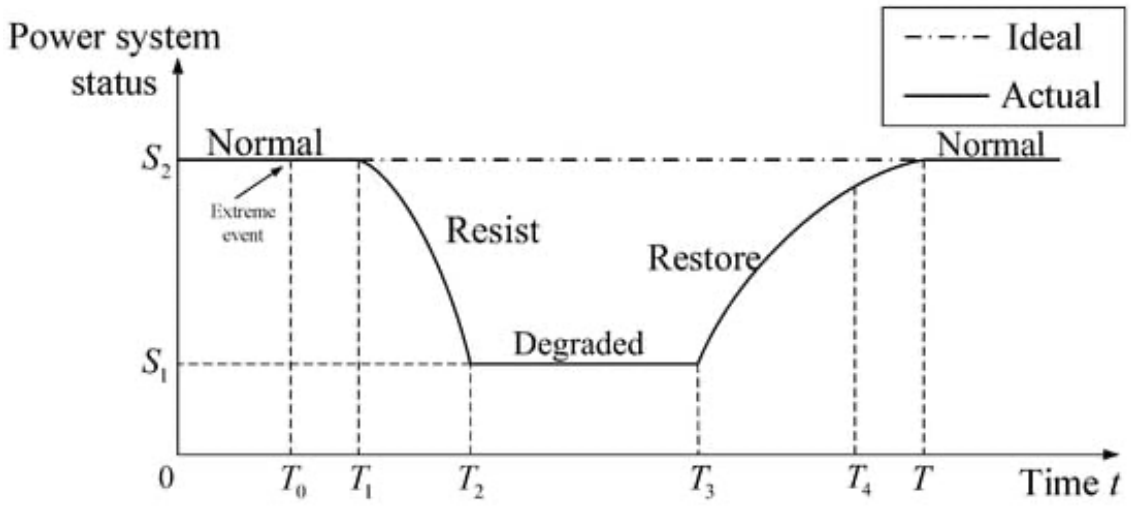



Figure 2.43 Resilience metric for a power system. Source: [13]

The figure presents how a metric can show the degradation of a system's performance over time during a not foreseen event, which can be a good selection for measuring resilience. In this case, the power system is degraded to level S1 and continuously restored to the normal state. If the degraded state below is shortened or prevented at all, the research can state that resilience has been enhanced [13].

### 2.11.1 Response Time

For software systems, response time is one of the most important metrics of performance. This metric not only reflects the performance of a system but also impacts the user experience. The user, who is the main consumer of the system, has a limited understanding of their background processes, which makes response time critical for the nature of the limited attention span [103].

This response time has not only a strong correlation with the system performance but also affects the revenue of a company such as Google, which reported

a 20% drop in traffic seen with a response time increase of 400ms [103]. Moreover, response time reduction reflects the enhancement of performance that can result in increased traffic and revenue. Different case studies, one such as Shopzilla in a year-long performance redesign, were able to reduce their response time from 7 to 2 seconds, increasing the traffic by 25% and a 7-12% revenue [104].

In conclusion, response time can be a good indicator of performance in a software system, and it can be used to measure its degraded states to observe if resilience has been enhanced or not.

### 2.11.2 Distributed Tracing

Distributed trace refers to the record of the paths taken by a single request and shows how it propagates throughout different systems [105]. Distributed tracing is widely an adopted technique to enable engineers to observe how distributed systems behave for debugging and optimization [106]. Without this valuable technique, pinpointing a performance bottleneck would be harder. This technique makes debugging and understanding of distributed systems less daunting and easier to improve [105].

Distributed tracing also covers some new issues that are only present in complex distributed systems. For example, complex distributed systems have an increased distance between the effect and cause. This means that an error in the distributed system can originate from a completely different place, it does not need to be at least related or have a low degree relationship. Another issue that is solved, is the inconsistency that comes due to the independence of services. For example, one service is completely independent of the other, but it can affect it, meaning that we cannot know how consistent changes and errors in a system will be [103].

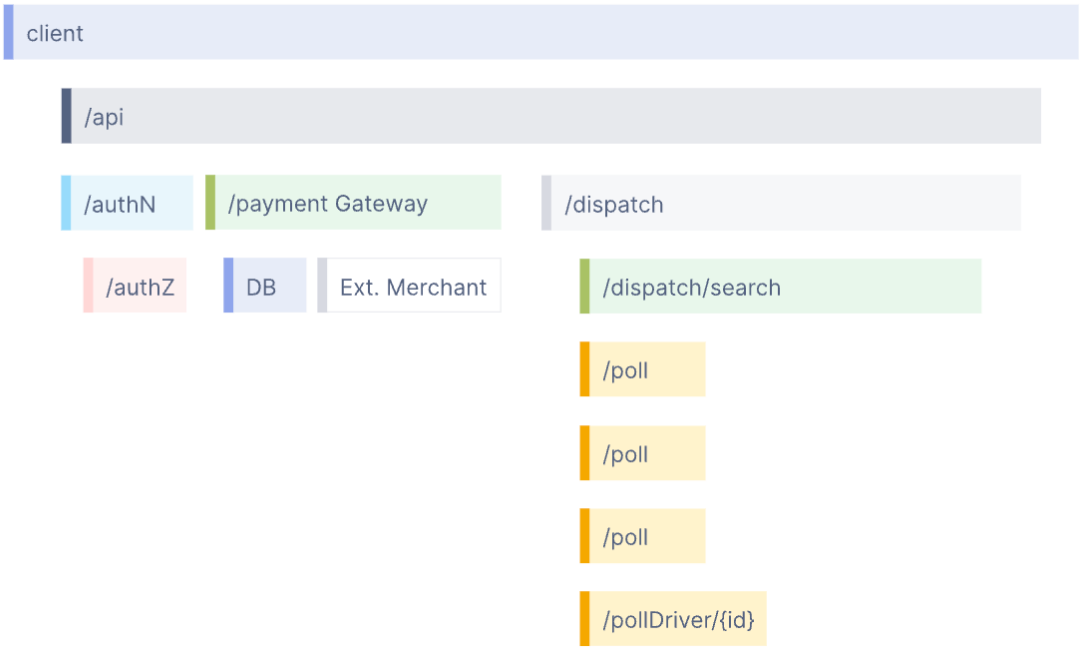



Figure 2.44 Waterfall diagram of a distributed transaction. Source: [105]

The previous image shows a waterfall diagram of the processes that make the main request made to the API. This example shows the parent and child relationship between each process. The AuthZ microservice is used by the AuthN, and so it is used by the API. This technique can reveal performance bottlenecks that can happen in very deep child processes.

### 2.11.3 Open Telemetry

OpenTelemetry is defined as an observability framework designed to create traces, metrics, and logs. It is vendor and tool-agnostic, meaning that a variety of different tools can be used [105]. OpenTelemetry is a standard that aims to provide a standardized way of cloud computing through the use of open format and production-ready binaries for cloud computer monitoring [23]. This standard can be used for collecting traces as metadata along with requests from other components in distributed systems, aka microservices [107].

OpenTelemetry has three major components, which are API, SDK, and a collector. These are interoperable and composable. The API and SDK are responsible for managing the span state and serializing logging data, while collectors are responsible for collecting data from the services [103].

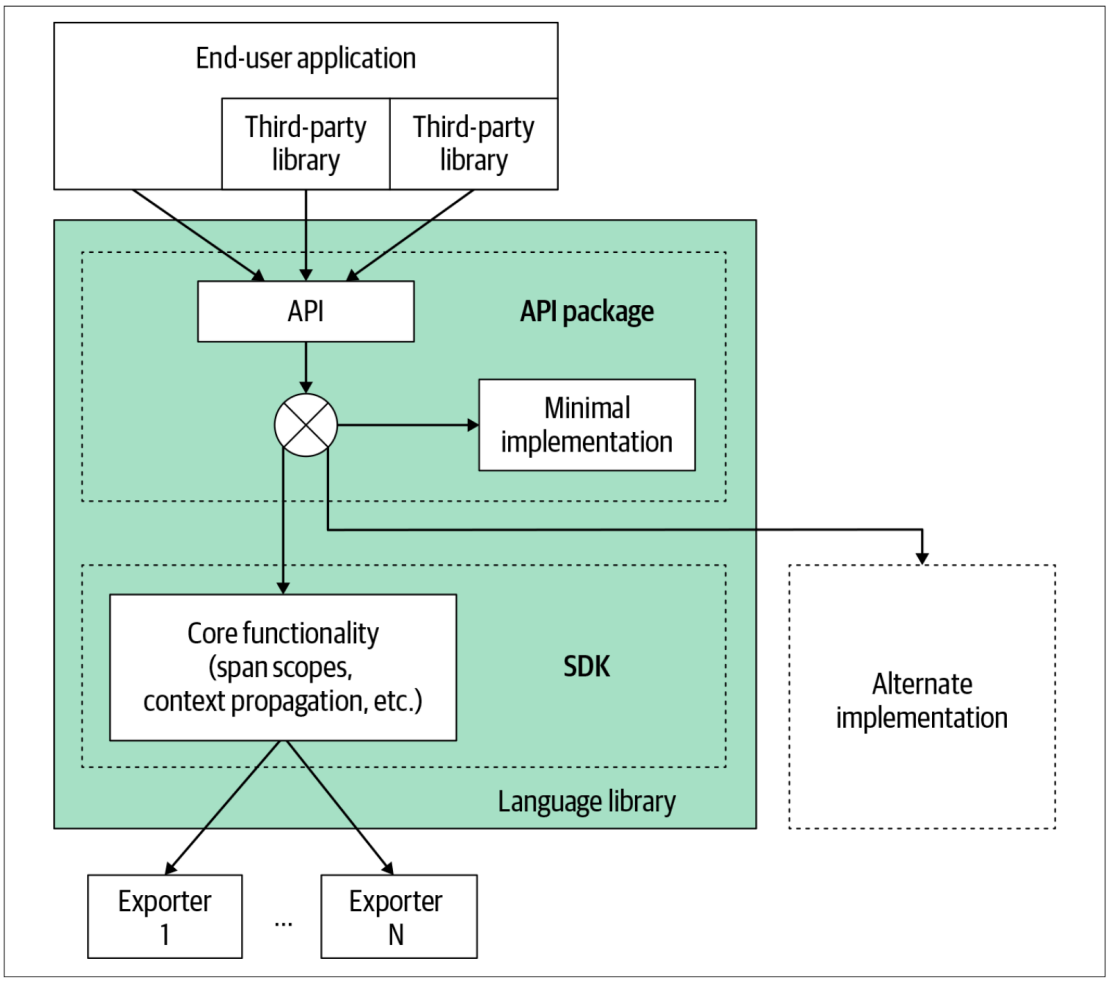



Figure 2.45 Diagram of OpenTelemetry architecture. Source: [103]

This figure shows the architecture and the main components of OpenTelemetry. The service sends the data into the API, which is later processed by the SDK to be finally exported into a different tool that can be used for visualization of the recollected data [103].

### 2.11.4 Jaeger

"A newer open source tracing tool and a Cloud Native Computing Foundation (CNCF) project. The more lightweight nature of Jaeger has made it extremely popular for cloud-native developers." [103].

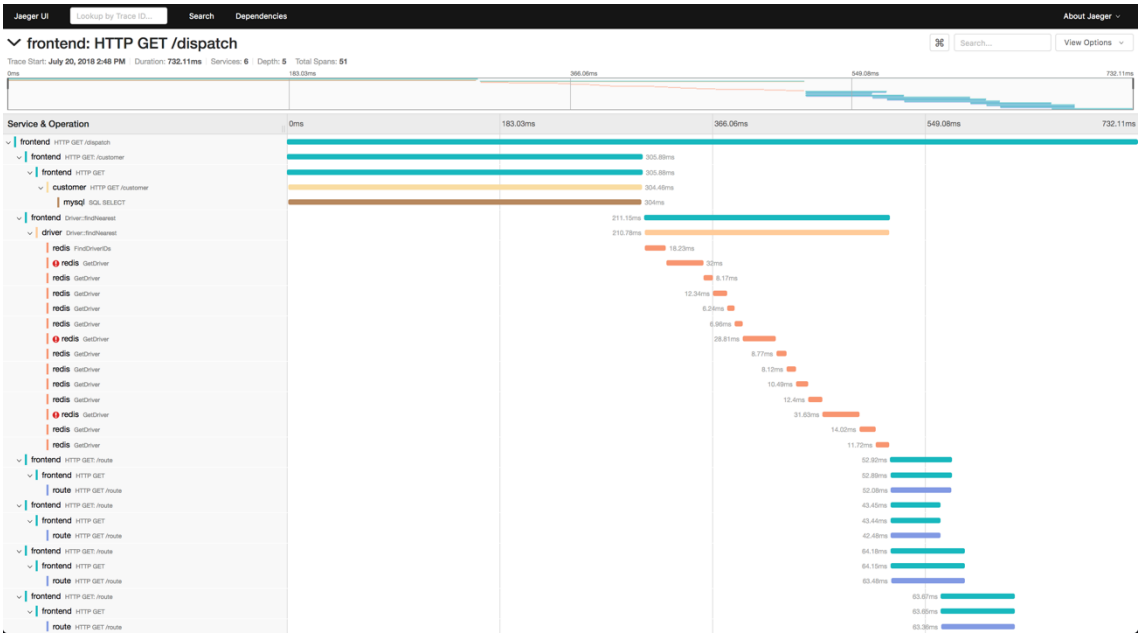



Figure 2.46 Trace Detail View Screenshot. Source: [108]

This image shows how Jaeger provided visualization of a detailed trace of a request. The trace details how the request propagates through the different processes, and how much time each one takes, exposing any possible performance bottlenecks.

### 2.11.5 Percentile

To have a base read of a metric is important to consider all the results of the set into account. Average, being a very common selection to get a value that represents a bigger set, sometimes cannot show the overall scenario. Averages can get skewed by outliers, which in software can become common due to transient faults such as network delays [103].

Percentiles can provide a better representation of a set. These can work great for performance metrics since they can represent a minimum service level agreement [103]. In the case of our selected metric, response time, percentiles can represent the maximum response time expected. This is applied to a percentage of 75%, 90% or even 99%.

| Request latency (ms) |
|---|
| 87 |
| 89 |
| 91 |
| 93 |
| 102 |
| 138 |
| 174 |
| 260 |
| 556 |
| 5000 |

| Selected statistics (ms) | 50th percentile |
|---|---|
| 120 | Average |
| 659 | 90th percentile |
| 1000 | Maximum |

Figure 2.47 Average and Percentile Source: [103]

This image shows the difference between an average and a percentile. The average might show the middle number of a set, which is not only affected by outliers but is also not a good representation of low and high values. On the other hand, the percentile shows the maximum value expected of a defined percent of the set. This means that here the 90 percentile (p90) result of 659ms, represents that there is a 90% probability that the request will be under 659ms.

# 3 Chapter 3: Methodology

The architecture of the system under test for this research shows the benefits of microservices architecture and the possible failures this architecture might present. With the use of the Chaos engineering discipline, we can develop experiments to test the overall resilience against specific scenarios, learn from them, mitigate them, and verify if the resilience is enhanced by measuring the performance metric, which is the response time percentile.

To research and validate what resilience patterns or practices enhance resilience, we need to follow the following steps, which are taken from the chaos engineering discipline. The following are:

1. Define a steady performance state based on metrics.
2. Define hypotheses of certain failure scenarios
3. Execute the experiment.

4. Detecting failures and enhancing resilience/reliability

## 3.1 Steady State Performance

As previously defined, why response time is a good performance metric for software systems, the steady-state or base performance of the distributed system under test is defined by a response time percentile of 90%. Percentile 90 is selected since the variation is not great under our system under test, and selecting a higher percentile would mean improving the 1% or 5% of performance in our system, making it harder to support or refute our hypothesis [21].

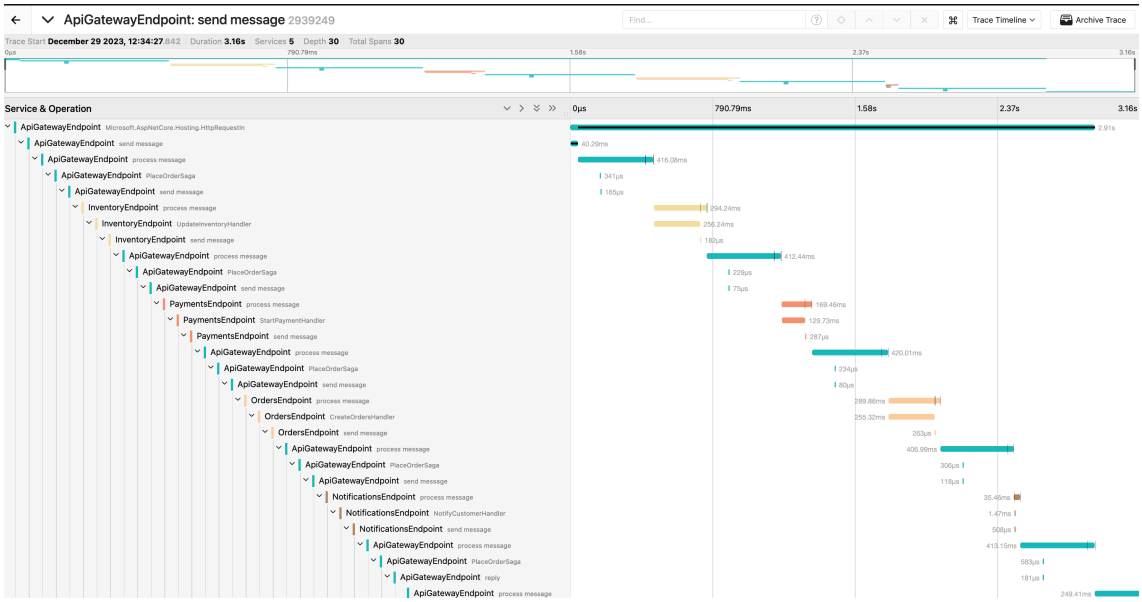



Figure 3. 1 Trace timeline. Source: own

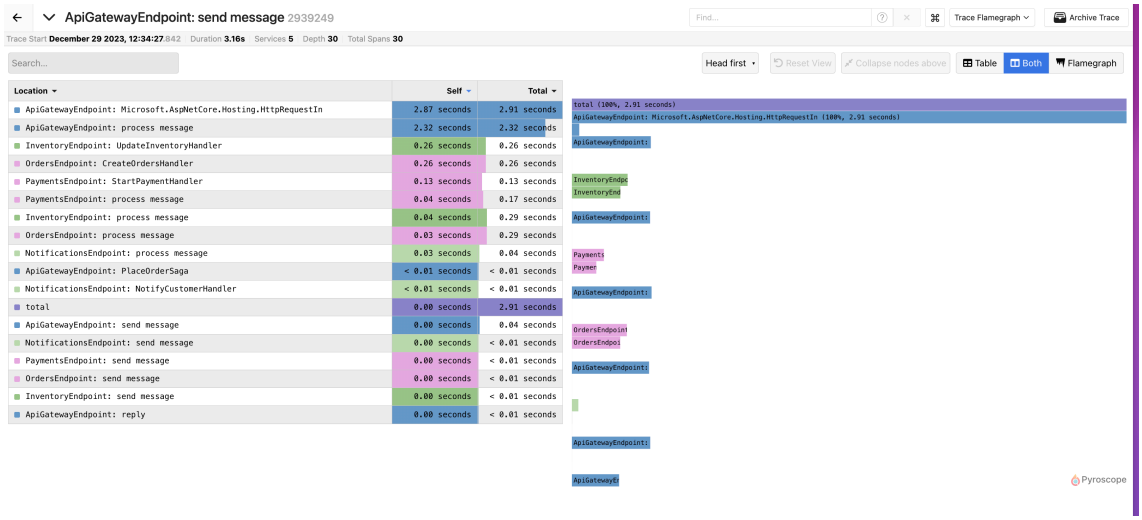



Figure 3. 2 Trace flame graph. Source: own

Our system under-test response time is measured by the Jaeger tool that leverages OpenTelemetry to provide a detailed trace of each request. The previous two images show a detailed visualization of the trace of a request that propagates throughout the many systems. The trace timeline shows how much time each service takes, exposing any bottleneck. Both traces show a response time of 3.16s.

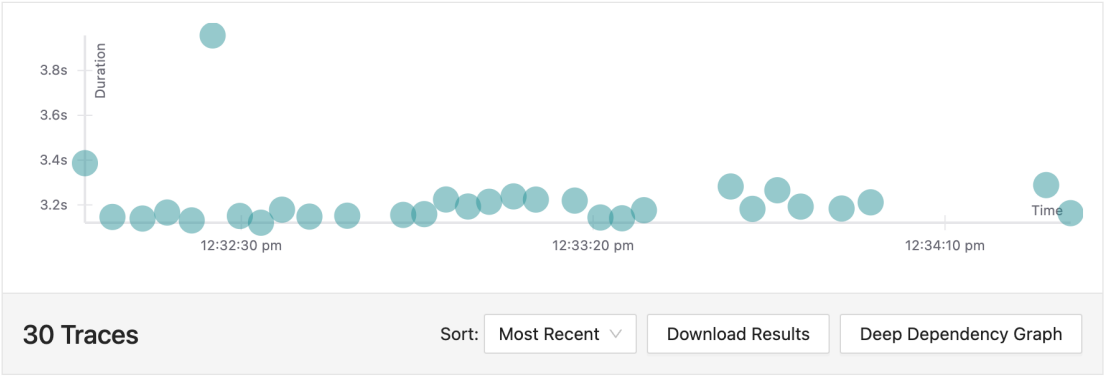



Figure 3. 3 Plot graph of response time. Source: own

The plot graph shows the duration of each request, also known as response time. Most of the requests take around 3.2s up to 3.3s with some outliers of 3.9s. The variance in the results is a very minimum of 0.02 which a big data set would provide a very similar representation of a small dataset.

| Response Time |
|---|
| 3,16 |
| 3,29 |
| 3,21 |
| 3,18 |
| 3,19 |
| 3,27 |
| 3,18 |
| 3,28 |
| 3,18 |
| 3,14 |
| 3,14 |
| 3,22 |
| 3,22 |
| 3,24 |
| 3,21 |
| 3,19 |
| 3,22 |
| 3,16 |
| 3,16 |

| |
|---|
| 3,15 |
| 3,15 |
| 3,18 |
| 3,12 |
| 3,15 |
| 3,96 |
| 3,13 |
| 3,17 |
| 3,14 |
| 3,15 |
| 3,39 |

Table 1 Base Response Time Source: Own

| 0% | 25% | Median | 75% | 90% | 100% | Variance |
|---|---|---|---|---|---|---|
| 3,12 | 3,15 | 3,18 | 3,22 | 3,29 | 3,96 | 0,022 |

Table 2 Response Time Percentile

The previous two tables show the data selected for the representation of the base steady state of performance of the system under test. The results of the percentile p75, and p90 are 3,22s and 3,29s accordingly.

## 3.2 Hypothesis

Before running experiments to test resilience patterns or other strategies, the definition of the hypothesis needs to be made, to confirm these strategies can enhance resilience. Because transient faults are different and unexpected, different types of experimentation need to be executed to test each resilience pattern or strategies.

### 3.2.1 Total outages in services

Many failures in the cloud could affect only a specific service that is part of a distributed system. This failure could include a complete failure of a microservice, that is crucial for the processing of a distributed transaction. In this case, the following hypothesis should be supported.

"If a microservice becomes unavailable, the response time will rise, because the system will need to wait the service to become available."

When a microservice fails, the request will not be able to be fulfilled, and the orchestrator will wait gracefully until a timeout has been reached and end the request as a failure. However, if resilience patterns mitigate this issue, as self-healing techniques of Kubernetes, with the synergy of retry patterns, the request could be completed with a higher response time. The following hypothesis should be supported with the previous resilience enhancements into account.

“If a microservice fails, then the request will timeout, because the system will not be able to fulfill the request.”

Finally, not every microservice has the same responsibility as others. There is the orchestrator which is responsible for keeping track of the saga state and the orchestration of steps of the distributed transaction. With persistence as a resilience pattern, the following hypothesis should be refuted.

“If the orchestrator service fails, then the saga process is stalled, because the system will not be able to persist the state of the process in memory.”

### 3.2.2 Network outages

Network outages can be triggered by any type of network phenomenon as high latency, firewall rules, network disruption, and many more, which is why it is important to experiment with a network outage case. The following hypothesis should be supported.

“If a network outage is introduced, then the response time will timeout because the system will not be able to fulfill the request in time.”

However, with retry patterns such as retry, fallback, or circuit breaker, the request could still be processed successfully with a higher response time. The following hypothesis should be supported.

“If a network outage is introduced, then the response time will rise because the system will need to retry the timeout process of the distributed transaction.”

### 3.2.3 Stress Degradation

Finally, services can degrade performance due to high stress on resources such as CPU and memory. This degradation by stress can be due to machine resources not being optimized, high traffic of requests, and many others. If one or more services have stress degradation in their performance, the following hypothesis should be supported.

“If a service presents degradation due to stress, the response time will rise, due to the need to retry timeout processes of the distributed transaction.”

## 3.3 Hypothesis validation and response time.

After the execution of experiments that best aim to test each hypothesis, the degradation of the system performance is measured. The measure of the performance of the system under test is response time, which is lower than 60 seconds. The default timeout of an HTTP request is 100 seconds in .NET [109] , however, it has been defined as 60 seconds, since our base time performance metric is 3,22s and 3,29 for p75 and p99 respectively.

During the refutation or support of the hypothesis, performance will be compared with each different resilience pattern or practice in order not to only confirm that resilience is enhanced, but also to what degree is enhanced.

# 4 Chapter: Results

This chapter presents the results provided by the experiments that best target the previously defined hypothesis about system performance during transient faults. Each experiment represents a certain type of common transient fault and discusses the cause of each result.

## 4.1 Pod or Service Failure Experiment

The experiment of Pod or Service failure simulates the outage of a specific service in the microservice architecture, which can introduce issues such as high response time, or even time-outs if the performance is affected to a higher degree. With this experiment, transient faults such as cloud provider failures, hardware failures, or even software crashes were simulated.

### 4.1.1 Pod Failure Experiment

This experiment aims at a specific pod, or service unit in Kubernetes. The experiment creates a failure in the pod, causing the service to become unavailable during a defined time.

Definition

```yaml
kind: PodChaos
apiVersion: chaos-mesh.org/v1alpha1
metadata:
  namespace: default
  name: podfailure
spec:
  selector:
    namespaces:
      - default
    labelSelectors:
      app: notification
  mode: all
  action: pod-failure
  duration: 30s
```

Figure 4.1 Pod Failure Experiment Definition. Source: own

This image shows the definition of the pod failure experiment. This experiment aims to make the notification service unavailable for 30s. After this, the notification service should become available.

```
(base) ➜  chaos.engineering git:(main) ✗ kubectl get pods -A
NAMESPACE       NAME                                        READY   STATUS    RESTARTS        AGE
cert-manager    cert-manager-5c9d8879fd-56jlk               1/1     Running   1               4d20h
cert-manager    cert-manager-cainjector-6cc9b5f678-nlrmv    1/1     Running   1               4d20h
cert-manager    cert-manager-webhook-7bb7b75848-6b28d       1/1     Running   1               4d20h
chaos-mesh      chaos-controller-manager-77fc976dbd-82w5x   1/1     Running   0               18m
chaos-mesh      chaos-controller-manager-77fc976dbd-cjhdb   1/1     Running   0               18m
chaos-mesh      chaos-controller-manager-77fc976dbd-fqn5g   1/1     Running   0               18m
chaos-mesh      chaos-daemon-gqc65                          1/1     Running   0               18m
chaos-mesh      chaos-dashboard-77f5cf9985-dv9gr            1/1     Running   0               18m
chaos-mesh      chaos-dns-server-779499656c-rvvng           1/1     Running   0               18m
default         inventory-deployment-5b9b7b7c6b-bhntc       1/1     Running   0               31m
default         notification-deployment-64d54d54db-gwqgh    1/1     Running   1 (30s ago)     31m
default         orchestratorapi-deployment-df49cd789-pts67  1/1     Running   0               31m
default         order-deployment-75cc96cb5c-l28d8           1/1     Running   0               31m
default         payment-deployment-5d4bb7cb87-v69hg         1/1     Running   0               31m
default         simplest-769f69ffd9-24mzx                   1/1     Running   0               31m
kube-system     coredns-5dd5756b68-5lsgt                    1/1     Running   3 (4d22h ago)   33d
kube-system     coredns-5dd5756b68-ff6k2                    1/1     Running   3 (4d22h ago)   33d
kube-system     etcd-docker-desktop                         1/1     Running   5 (4d22h ago)   33d
kube-system     kube-apiserver-docker-desktop               1/1     Running   5 (4d22h ago)   33d
kube-system     kube-controller-manager-docker-desktop      1/1     Running   5 (4d22h ago)   33d
kube-system     kube-proxy-twfpn                            1/1     Running   3 (4d22h ago)   33d
kube-system     kube-scheduler-docker-desktop               1/1     Running   5 (4d22h ago)   33d
kube-system     storage-provisioner                         1/1     Running   7 (4d3h ago)    33d
kube-system     vpnkit-controller                           1/1     Running   3 (4d22h ago)   33d
observability   jaeger-operator-746fcb5c87-64nfv            2/2     Running   2               4d21h
```

Figure 4.2 Pod Failure Experiment Status. Source: own

This image shows the running pods in the selected cluster. The notification pod is aimed at the experiment, so Kubernetes keeps restarting it in order to bring it to an operational state. During the 30 seconds that the experiment lasts, the container under the notification pod is not working properly.

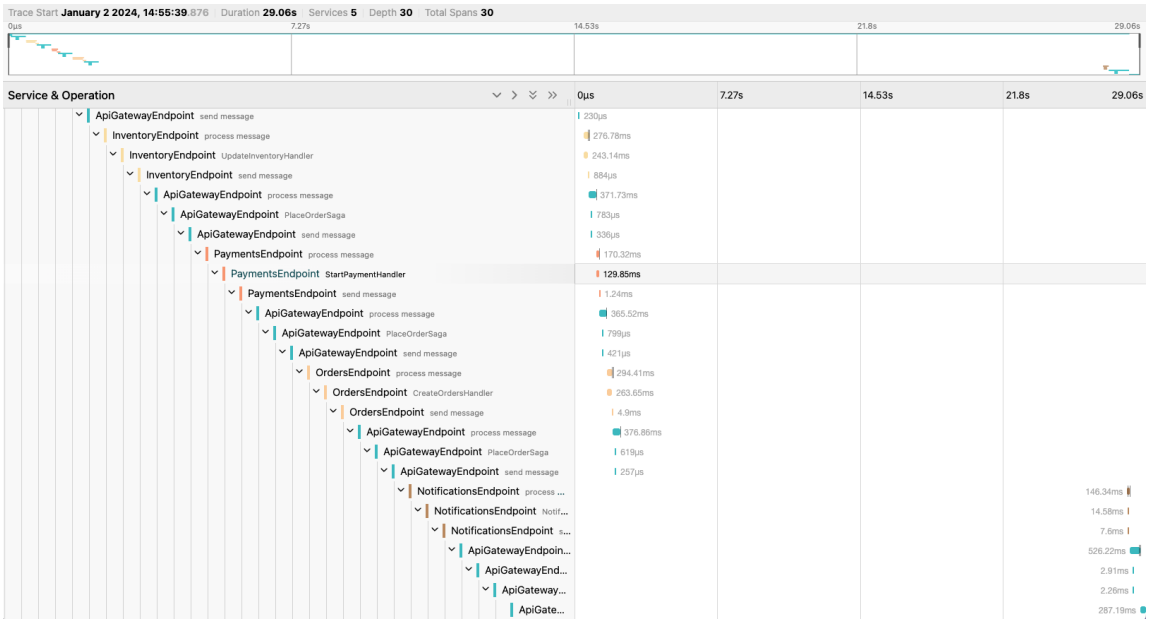



Figure 4.3 Detailed Trace for Pod Failure Experiment. Source: own

This image shows a trace of a request that was executed during the experiment. The request here has a response time of 29.06s. This is a clear increase in the response time of 802.48% and 783.04% for the percentile 75 and 90 accordingly to the performance of the application. The request was executed a few seconds after the start of the experiment, which is why the request takes no longer than the duration of the experiment since Kubernetes self-healing mechanism restores the service as soon as the chaos engineering experiment is finished.

### 4.1.2 Pod Kill Experiment

The pod kill experiment deletes a pod. Leaving the system incomplete and unable to process properly a request. This experiment simulates events such as accidental deletion.

Definition

```yaml
kind: PodChaos
apiVersion: chaos-mesh.org/v1alpha1
metadata:
  namespace: default
  name: podkill
spec:
  selector:
    namespaces:
      - default
    labelSelectors:
      app: notification
  mode: all
  action: pod-kill
  duration: 30s
  gracePeriod: 1
```

Figure 4.4 Pod Kill Experiment Definition. Source: own

This image shows the declaration of an experiment that aims to kill an instance of a pod in Kubernetes. The experiment aims for the notification service.

```
(base) ➜  chaos.engineering git:(main) ✗ kubectl get pods
NAME                                        READY   STATUS    RESTARTS        AGE
inventory-deployment-5b9b7b7c6b-99ktz       1/1     Running   3 (8m25s ago)   8m56s
notification-deployment-64d54d54db-gdvrj    1/1     Running   0               3s
orchestratorapi-deployment-df49cd789-bfkkz  1/1     Running   3 (8m25s ago)   8m56s
order-deployment-75cc96cb5c-8z8fm           1/1     Running   3 (8m33s ago)   8m56s
payment-deployment-5d4bb7cb87-59qtl         1/1     Running   3 (8m31s ago)   8m56s
simplest-769f69ffd9-5wq8v                   1/1     Running   0               8m47s
(base) ➜  chaos.engineering git:(main) ✗
```

Figure 4.5 Pod Kill Experiment Status. Source: own

During the monitoring of the experiment, Kubernete's self-healing mechanism takes care of the missing pod, which was killed, and restores it as soon as possible. The commands show each pod and the age that it has since it was started. Notification pod has an age of 3s, just a few seconds after it was killed by the chaos engineering experiment. Fortunately, the system performance is not affected since Kubernetes brings back the pod to life.

### 4.1.3 Third-Party Service Failure Experiment

Third-party services are common in any type of system. These can help with payments, notifications, localization services and more. Even though most of these services are reliable, they can present unforeseen failures, that the system is not prepared to handle. This experiment consists of a manual injection of unhandled exception in the payment service.

```
fail: NServiceBus.MoveToError[0]
      Moving message '7eae9e26-e83f-4395-aadc-b0eb00f673c6' to the error queue 'error' because processing failed due to an exception:
      System.Exception: UNHANDLED EXCEPTION
         at Payments.Microservice.Handlers.StartPaymentHandler.Handle(StartPayment message, IMessageHandlerContext context) in /Users
         at NServiceBus.InvokeHandlerTerminator.Terminate(IInvokeHandlerContext context) in /_/src/NServiceBus.Core/Pipeline/Incoming
         at NServiceBus.LoadHandlersConnector.Invoke(IIncomingLogicalMessageContext context, Func`2 stage) in /_/src/NServiceBus.Core
         at NServiceBus.MutateIncomingMessageBehavior.InvokeIncomingMessageMutators(IIncomingLogicalMessageContext context, Func`2 ne
         at NServiceBus.DeserializeMessageConnector.Invoke(IIncomingPhysicalMessageContext context, Func`2 stage) in /_/src/NServiceB
         at NServiceBus.UnitOfWorkBehavior.InvokeUnitsOfWork(IIncomingPhysicalMessageContext context, Func`2 next) in /_/src/NService
```

Figure 4.6 Third-party or unhandled exceptions experiment. Source: own

Since Chaos Mesh, cannot influence a failure of a third-party app, this test is done manually to include an unhandled exception. The image shows how the NServiceBus framework manages this, by moving them to an error queue, to retry or keep a log of it.

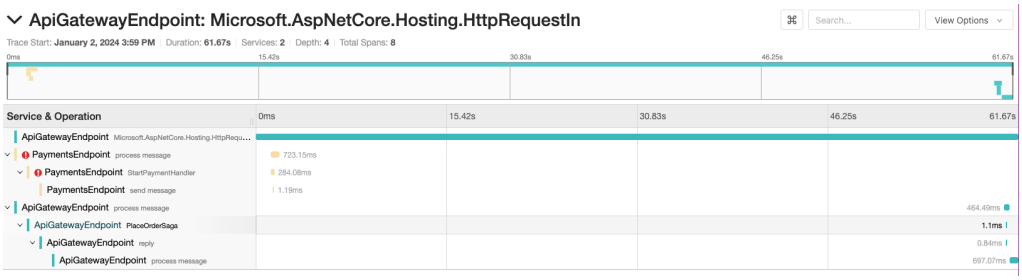



Figure 4.7 Detailed Trace for Third-Party Failure Experiment. Source: own

The detailed trace shows a response time of 61.67s which usually means the maximum of the request due to the configured time-out pattern, which NServiceBus natively implements. The trace shows that the error is on the payment service. The results of an increase in response time are 1,815.22% for the percentile 75 and 1,775.04% for the percentile 90. This specific scenario will not benefit directly in a reduction of response time, but in the proper process of a request since this has not been processed successfully, and has terminated the request at a maximum time of 60 seconds.

## 4.2 Network Outage Experiment

This experiment simulates the transient faults of communication between the services. This can be expected in any service, even in third-party services and on the message queue that communicates all the services.

**Definition**

```yaml
kind: NetworkChaos
apiVersion: chaos-mesh.org/v1alpha1
metadata:
  namespace: default
  name: loss90all
spec:
  selector:
    namespaces:
      - default
    labelSelectors:
      app: payment
  mode: all
  action: loss
  duration: 5m
  loss:
    loss: '90'
    correlation: '0'
  direction: to
```

Figure 4.8 Network experiment definition. Source: own

The image shows the definition of a network outage experiment. Even though Network Chaos can cause delays, high packet loss rate, packet reordering and more, packet loss represents better a network outage, such as high bandwidth usage, high latency, interference of physical connections, and more. The experiment targets all services for 5 minutes with a 90% loss to make the experiment more severe.

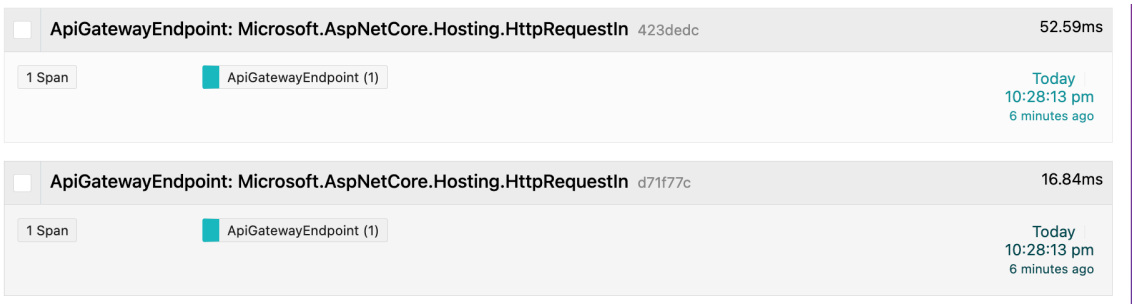



Figure 4.9 Network experiment result. Source: own

The experiment can create a complete outage in the system. The system is not able to follow any following steps after the initial HTTP entry point. Two of the requests are registered here, and none of them publish a message queue for the next step. None of the requests were fulfilled or started.

## 4.3 Stress Failures

The stress failure experiment simulates events such as an unexpectedly high load in the system, a non-optimized application, or poor resource management. The chaos engineering experiment created processes that consume resources of a specific pod.

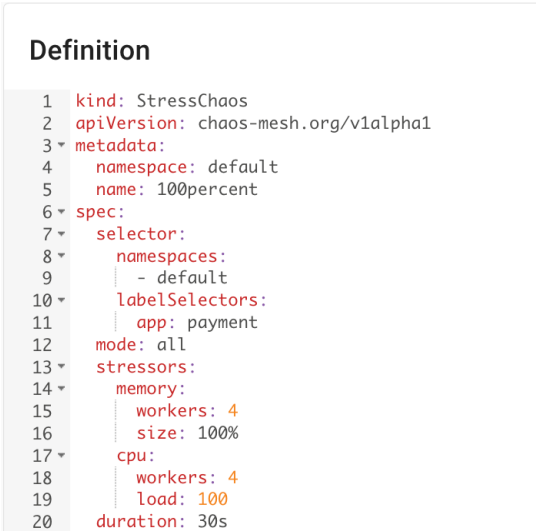



Figure 4.10 Stress experiment result. Source: own

The image shows the definition of the stress experiment that targets the payment service. The experiment sets 4 workers in the CPU stressor of the service with 100% of usage per worker and a load of 100MB in the memory. The experiment lasts 30 seconds, enough to make a request and create a detailed trace that exposes how each service performance is.

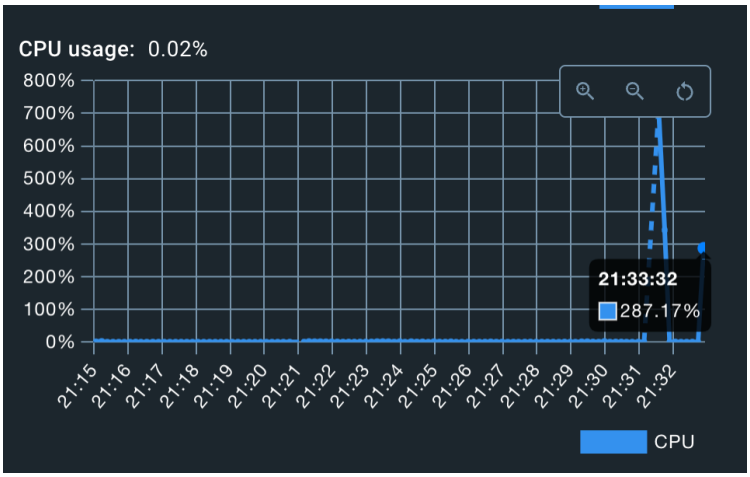



Figure 4.11 Stress experiment CPU monitor. Source: own

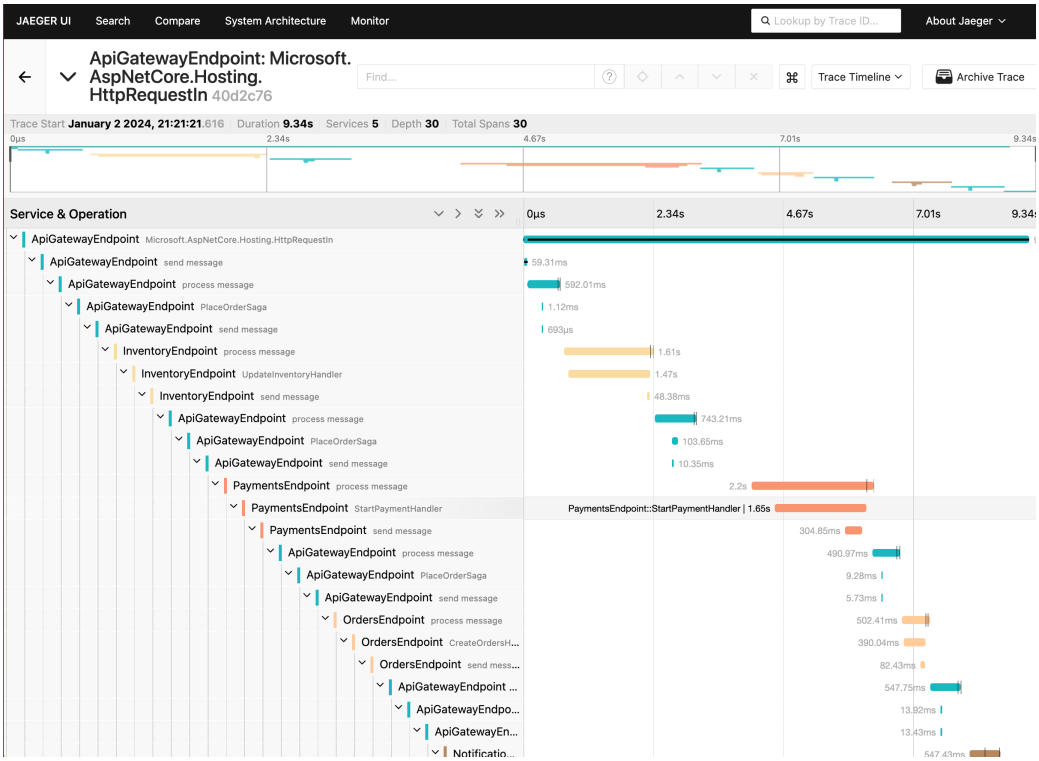



Figure 4.12 Stress experiment Trace detail. Source: own

During the defined time for the execution of the experiment, the CPU of the selected payment services presents a spike of 287% in its CPU usage. This percentage greater than 100% represents how the chaos experiment is targeting the virtual machine, instead of the container, meaning that the CPU usage in the virtual host is greater than the container [110]. The detailed trace shows a response time of 9.34s which is an increase of 190.06% and 183.98% for the percentile 75 and 90 respectively.

# 5 Chapter: Discussions

As previously seen, the three experiments run to simulate service outages presented different outcomes depending on the type of the experiment. Each experiment, unique by itself, was executed in a controlled Kubernetes cluster, with Jaeger as the observability tool.

## 5.1 Discussion of Service Outages

### 5.1.1 Container Failure Experiment

The first experiment, aimed to simulate a container failure, which can be caused by different events. A container failure can be also called a container crash. They refer to the failure of services that run inside a container, and they are so common that many studies have focused on automation for the detection, reporting, and fix of these crashes [111].

The most common reasons these container crashes happen can be because of updates in the software, bugs in the software, improper configuration, unhandled exceptions, and more. A container crash can even be triggered by a network timeout that prevents the container from replying to the health monitoring requests, meaning that the container orchestration tool, in this case, Kubernetes, will treat this container as unhealthy and restart it.

During the first experiment of container failure, the chaos controller responsible for running the experiment injected a failure in the notifications service. Here the long-term running saga kept waiting for the failing service to respond. This is possible due to the uncoupled architecture of the message broker, where the message is relayed from the orchestrator and kept by the message broker until the services are restored.

Should the system architecture not implement communication between services with message brokers, but with common HTTP requests, the requests would have been sent to the failed service and not processed correctly. This would mean that the notification service would have not known about the request sent previously by the orchestrator, and the orchestrator would be forced to terminate the request in a timeout.

### 5.1.2 Container kill experiment.

The second experiment is more drastic than the previous one. This experiment deletes a complete pod from the system infrastructure. This means that not only the pod it not available, but it does not exist. This experiment can seem identical to the previous one, but it achieves different results, providing new insights of the system.

During the second experiment of container kill, the chaos controller aims the notifications service and deletes its instance. However, the obtained results, make it seem that there were no changes and that no pod was deleted. This happens due to the self-healing mechanism of the Kubernetes. Kubernetes, as soon as it detects a pod that is previously defined in the deployment, does not exist, it starts a new instance. That is why, a pod kill in the system under test infrastructure, is healed instantly by Kubernetes.

Should the system not use a container orchestrator, or implement any self-healing techniques for its infrastructure, the failure would become imminent since the orchestrator service would keep waiting for the notification service that does not exist and be forced to terminate the request in a timeout. To fix this issue without a container orchestrator that automates the monitoring, and management of containers, the human factor would be necessary, consuming resources and losing precious time.

### 5.1.3 Third-party Service Outage Experiment

The third experiment differs greatly from the two previous one. It is still a service outage type experiment, but it aims for a third-party service. To make it more drastic, an unhandled exception was injected manually into the payments service, to simulate an unexpected outage.

During the experiment, the orchestrator service treated the requests as a timeout since the payment step was not completed successfully. This case applies the rollback logic applied by the saga pattern, where a rollback of the committed changes is done, to keep data consistency.

Should the system apply the retry resilience pattern, the unsuccessful request to the third-party service could be retried in an immediate or delayed manner to reduce error probability. The system could also apply a circuit breaker and fallback pattern to use a second third-party service that is available, should the first third-party service keep failing.

### 5.1.4 Experiments Hypothesis

After the discussion of how the experiments affected the system infrastructure, and what happened when it did not, the following results to the hypothesis are supported.

1. “If a microservice becomes unavailable, the response time will rise, because the system will need to wait for the service to become available.”
2. “If a microservice fails, then the request will timeout, because the system will not be able to fulfill the request.”

These hypotheses are supported by the previous experiments; however the second can be refuted with the previous resilience patterns discussed.

Finally, this hypothesis is refuted since the previous experiments showed that the container orchestrator keeps at least one instance running, and our Saga framework, NServiceBus, implements in a native manner data persistence for the long-running processes, also known as sagas.

1. “If the orchestrator service fails, then the saga process is stalled, because the system will not be able to persist the state of the process in memory.”

## 5.2 Discussion of Network Outages

For the experimentation of networking, a single experiment was executed that simulated a network outage. Even though, different phenomena can cause a network outage, the same effect will be caused, a network outage or unavailability to communicate services throughout the network.

Networking transient faults can manifest unpredictably and frequently without being accurately identified and comprehended, which affects the reliability and safety of any system [112].

The unique network outage experiment created package loss through the use of the NET_SCH_NETEM module in the Linux kernel [86]. This experiment lasted 5 minutes and created a 90% packet loss. In other words, the service would need to retry to send a message through the network 10 times to get one successful request. The experiment affected all the running pods, meaning that all services would fail during the experiment.

During the experiment, the trace provided insight into how the request was not able to propagate to the desired systems. This happened in the first step of the saga, the orchestrator tries to send a message to the message broker to relay the next step to the designed next microservice, however due to the packet loss of the network outage, the service is unable to publish this message with the RabbitMQ connection.

Should the system apply resilience patterns and practices, such as retry patterns, or persistence of the long-running process, it would have been able to fulfill these requests despite the packet loss caused by the network outage. With the retry pattern, the distributed system would be able to retry and get the 10 packets required for one to succeed. Then, with the persistence of sagas, when a successful message arrived, the saga state would update; without the limitation of a timeout, if defined, the saga would complete even though the performance would have not been optimal.

### 5.2.1 Experiments Hypothesis

After the discussion and analysis of the network outage experiment, the following hypothesis can be supported:

1. “If a network outage is introduced, then the response time will timeout because the system will not be able to fulfil the request in time.”

The following hypothesis is refuted:

1. “If a network outage is introduced, then the response time will rise because the system will need to retry the timeout process of the distributed transaction.”

## 5.3 Discussion of resource stress Outages

Resources are not unlimited, which means that they always are a limiting factor in a system’s performance and scalability. With the use of containerization in the

system under test, a single cluster or virtual machine is used, sharing resources, for the different microsystems. The nature of containerization gives many advantages with the responsibility of system resource management.

The resources stress experiment seeks to simulate transient faults in the resources of a specific system. These transient faults can be caused by a high sudden load, a system bug, and many others.

During the experiment, the payment microservice is aimed for 30 seconds. The detailed trace shows a higher processing time than normal, even though the service simulates a third-party service consumption, there is no wait time for the third party. The microservices resources are limited and they are placed under stress that the application base system, needs to handle the other processes before the fundamental application processes, translating into a higher processing time.

Should the system apply resilience patterns and practices as bulkhead patterns, the resources of another process would have not been affected. Another way to tackle this stress issue is to implement auto-scaling of the services, or the automatic horizontal scaling feature of Kubernetes as a bulkhead pattern strategy. When the automatic horizontal scaling feature is implemented, the system will automatically create a new stateless instance of the microservice in stress, making the system able to process the request.

### 5.3.1 Experiments Hypothesis

After the discussion and analysis of the results of the resources stress experiment, the following hypothesis can be supported:

1. “If a service presents degradation due to stress, the response time will rise, due to the need to retry timeout processes of the distributed transaction.”

## 5.4 Discussion of the Ideal High Resilience System

For an ideal high-resilience system, only the required resilience pattern and practices must be implemented. Implementing more patterns than necessary can enhance resilience against more unforeseen transient patterns but will increase unnecessary complexity for this research.

A handful of effective patterns that will increase resilience are implemented natively by NServiceBus, others manually, and others by Kubernetes. After gaining insight into the system under test through the chaos engineering experiments, the following enhancements have been made in order to enhance its resilience in the previous scenarios.

### 5.4.1 Enhancing the System with Persistence.

The persistence pattern, required by NServiceBus for sagas, provides the persistence of a saga state in a data store. In this research, MongoDB is selected for its high availability feature previously discussed.

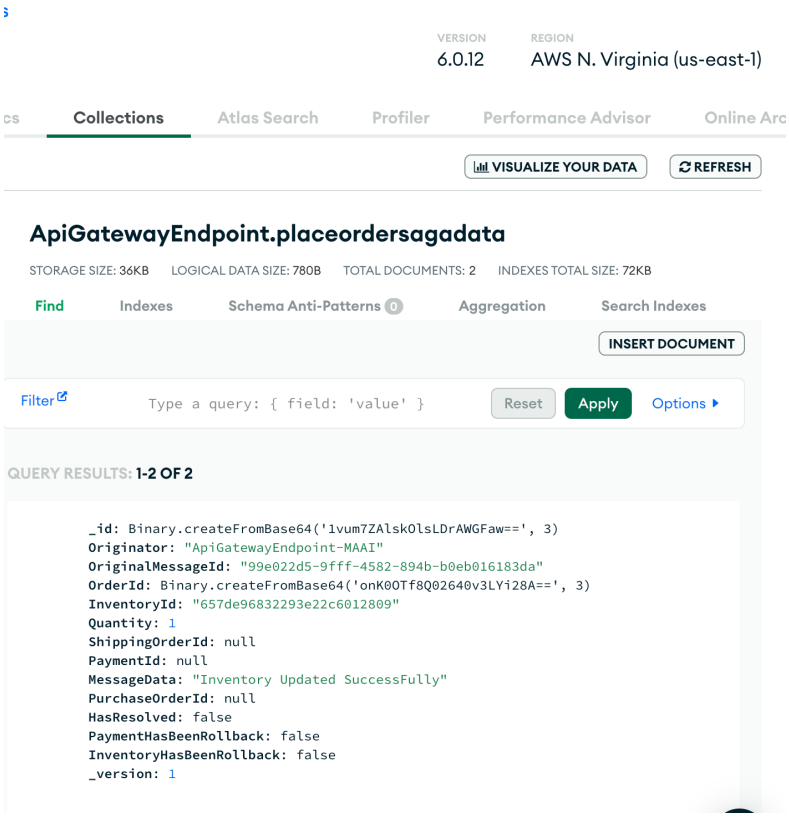



Figure 5.1 Saga data state persistence in MongoDB. Source: own

The image shows the data of the saga state, saved in a persisted data store. This means that any microservice or even the orchestrator, does not need to keep track of the state and can be restarted due to failure or management and will not create a problem. The saga state data is modeled uniquely per each different process, the 'PlaceOrderSagaData' defines properties such as inventory Id, Shipping Order Id, and Purchase Order Id, necessary to identify which record in the database needs to be rollback in case of an error. It also defines two rollback flags, which identify which step has been successfully rollback since each one is required only at a certain step of the process. This enhancement will tackle the failure of the orchestrator, or the network outage failure.

### 5.4.2 Enhancing the System with Retry Pattern.

The next enhancement is enhancing resilience through the use of the retry pattern, which can be natively implemented by NServiceBus. NServiceBus captures any unhandled exception and retries it immediately or in a delayed manner. The aim of the recoverability that is built in in NServiceBus is to tackle issues such as third-party service outages, such as the one demonstrated in the chaos engineering experiment, or other transient events such as database deadlock.

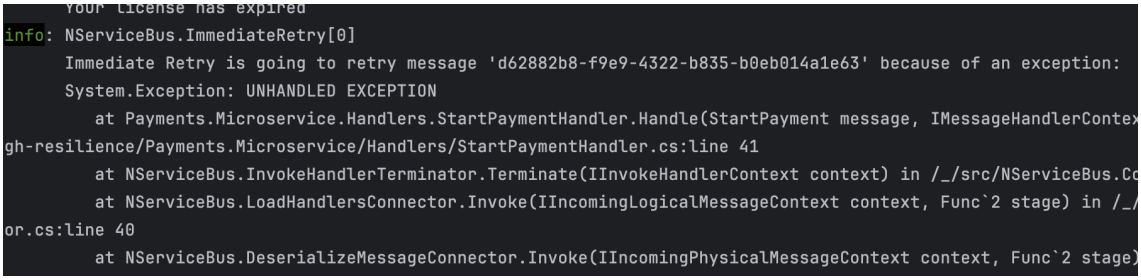



Figure 5.2 Retry pattern in NServiceBus. Source: own

The image shows how the 'UNHANDLED EXCEPTION' introduced in the third-party service was retried immediately. Because this is a manually introduced exception, the immediate retry will fail, but in a real-life scenario, a higher probability will come with a retry of a third-party service.

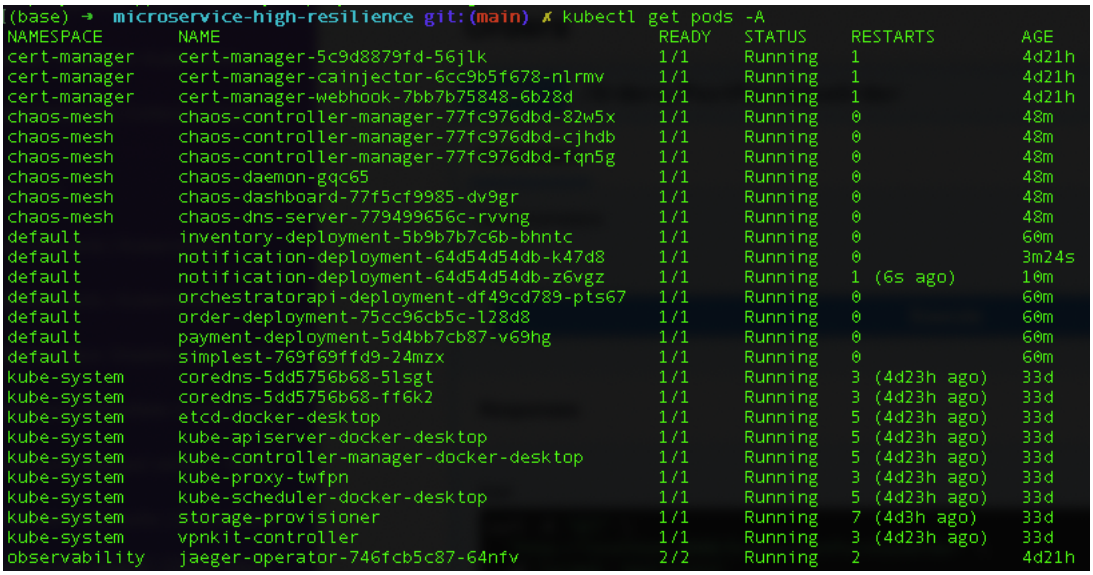
### 5.4.3 Enhancing System with Circuit Breaker.

In case the built-in strategies of the retry pattern do not solve the issue, NServiceBus manages an error queue in order to manage errors as the system has previously defined. NServiceBus applies an automatic circuit breaker strategy that switches a microservice into a processing mode, which retries each message in a delayed manner and waits for it to be successfully processed to take the next one.

### 5.4.4 Enhancing System with Automatic Scaling.

With the insight gained in the pod failure experiment, the use of automatic scaling was a resilience enhancement opportunity. Taking advantage of the declarative approach and self-healing feature of Kubernetes, the notification service, which is aimed during the pod failure experiment, now presents a second instance, that will be able to handle requests while the main instance presents a failure.

The same solution can work also for the resource stress experiment. Kubernetes support automatic horizontal scaling based on resource parameters such as CPU and memory. With the use of the message broker pattern, a new instance could load a balance request or takeover when a primary presents failure.



```
(base) ➜  microservice-high-resilience git:(main) ✗ kubectl get pods -A
NAMESPACE       NAME                                         READY   STATUS    RESTARTS         AGE
cert-manager    cert-manager-5c9d8879fd-56jlk                1/1     Running   1                4d21h
cert-manager    cert-manager-cainjector-6cc9b5f678-nlrmv     1/1     Running   1                4d21h
cert-manager    cert-manager-webhook-7bb7b75848-6b28d        1/1     Running   1                4d21h
chaos-mesh      chaos-controller-manager-77fc976dbd-82w5x    1/1     Running   0                48m
chaos-mesh      chaos-controller-manager-77fc976dbd-cjhdb    1/1     Running   0                48m
chaos-mesh      chaos-controller-manager-77fc976dbd-fqn5g    1/1     Running   0                48m
chaos-mesh      chaos-daemon-gqc65                           1/1     Running   0                48m
chaos-mesh      chaos-dashboard-77f5cf9985-dv9gr             1/1     Running   0                48m
chaos-mesh      chaos-dns-server-779499656c-rvvng            1/1     Running   0                48m
default         inventory-deployment-5b9b7b7c6b-bhntc        1/1     Running   0                60m
default         notification-deployment-64d54d54db-k47d8     1/1     Running   0                3m24s
default         notification-deployment-64d54d54db-z6vgz     1/1     Running   1 (6s ago)       10m
default         orchestratorapi-deployment-df49cd789-pts67   1/1     Running   0                60m
default         order-deployment-75cc96cb5c-l28d8            1/1     Running   0                60m
default         payment-deployment-5d4bb7cb87-v69hg          1/1     Running   0                60m
default         simplest-769f69ffd9-24mzx                    1/1     Running   0                60m
kube-system     coredns-5dd5756b68-5lsgt                     1/1     Running   3 (4d23h ago)    33d
kube-system     coredns-5dd5756b68-ff6k2                     1/1     Running   3 (4d23h ago)    33d
kube-system     etcd-docker-desktop                          1/1     Running   5 (4d23h ago)    33d
kube-system     kube-apiserver-docker-desktop                1/1     Running   5 (4d23h ago)    33d
kube-system     kube-controller-manager-docker-desktop       1/1     Running   5 (4d23h ago)    33d
kube-system     kube-proxy-twfpn                             1/1     Running   3 (4d23h ago)    33d
kube-system     kube-scheduler-docker-desktop                1/1     Running   5 (4d23h ago)    33d
kube-system     storage-provisioner                          1/1     Running   7 (4d3h ago)     33d
kube-system     vpnkit-controller                            1/1     Running   3 (4d23h ago)    33d
observability   jaeger-operator-746fcb5c87-64nfv             2/2     Running   2                4d21h
```

Figure 5.3 Replica set of notification Service in Kubernetes. Source: own

The image shows all the running containers in the cluster used for testing. Notification service now presents two instances, one is under the pod failure experiment, but the other is available for request handling.

### 5.4.5 Enhanced System Results.

With the enhancements done to the system under test, the same experiments were executed again, in order to provide insights in the handling of the requests.

Figure 5.4 Enhanced system trace detail with pod failure experiment. Source: own

After the resilience enhancements to the system under test, the pod failure experiment was executed. gain, and a request was triggered. The trace details show the result of 3.07s which is an improvement in performance of 4.66% and 6.66% for the percentile 75 and 90 respectively.

A performance improvement is seen, due to the extra instance of a service, meaning that more resources are allocated for the management of the requests, even though some are under test.

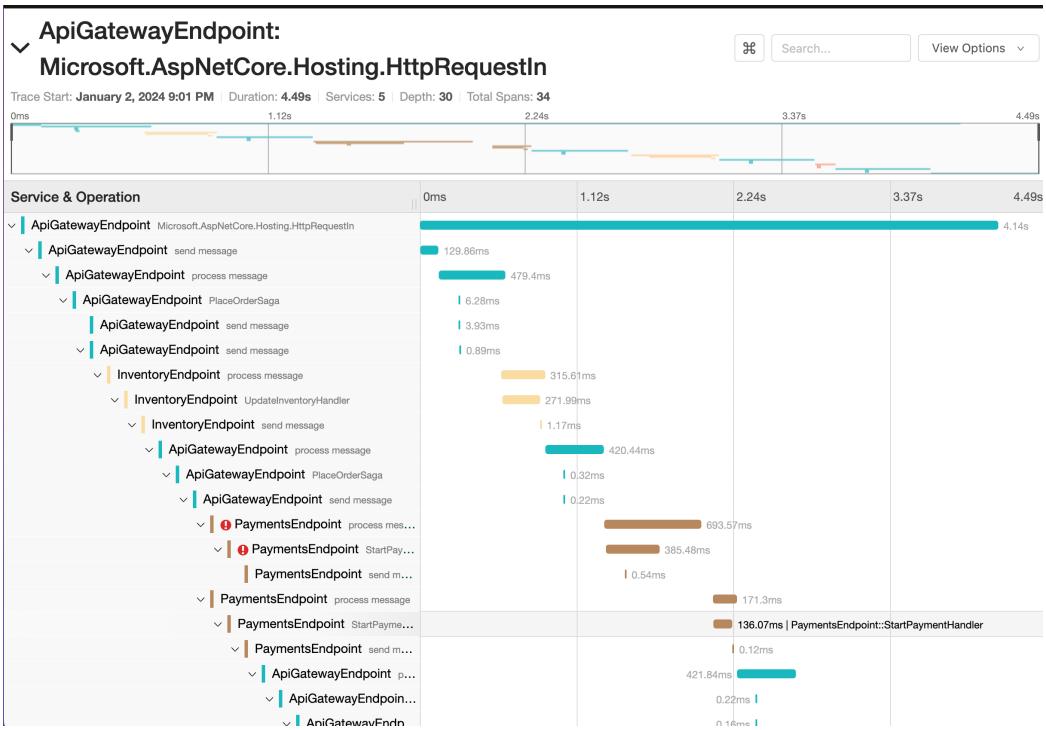



Figure 5.5 Enhanced system trace detail with third-party outage experiment. Source: own

The image shows the outcome of a request with the enhancements made. Here the request was finished in 4.49s, meaning 39% and 36.52% for the percentile 75 and 90 accordingly. The trace shows how the payments endpoint presents an error but is retried immediately and succeeds the second time.

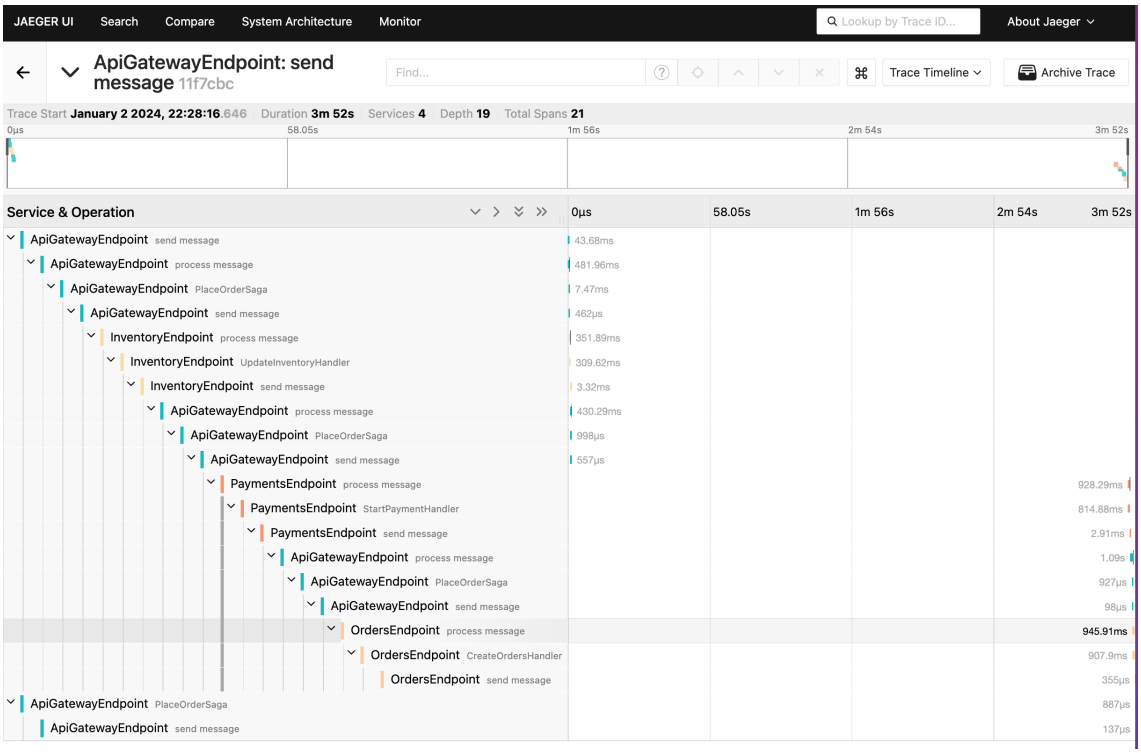


Figure 5.6 Enhanced system trace detail with network outage experiment.
Source: own

The detailed trace shows the propagation of a completed trace during the experimentation of the network outage. Here the results show a response time of 232 seconds, or 3 minutes and 52 seconds. Even though this is a higher time of completion than the timeout limit, the request is completed due to the circuit breaker pattern that moves the services in the delayed requests.

# 6 Chapter: Implication and Future Results

## 6.1 Conclusion

Our main hypothesis states that resilience can be enhanced through the use of resilience patterns in the saga pattern for distributed transactions. For the validation of this hypothesis, base performance was defined before the experiments that simulated transient faults. These experiments enabled insights into the weakness of the system against these transient failures and enabled the design of a more robust resilient system through the use of specific patterns and practices.

After the enhancements, experiments were executed again to capture performance metrics.

| Experiment Type | Pre-enhacements | Post-enhacements |
|---|---|---|
| Network Outage | NA | 232 |
| Third Party Failure | 61,67 | 39,44 |
| Pod Failure | 29,06 | 3,39 |

Table 3 Results of experiments before and after enhancements

The previous results collected before and after enhancements while running experiments show that performance is enhanced and in some scenarios, requests are completed successfully in a high response time instead of timing out.

With these metrics, we can support our hypothesis stating that the saga pattern for distributed systems can be enhanced by applying resilience patterns and practices.

## 6.2 Limitation of Chaos Engineering Experiments

Chaos Engineering experiments, even though provide valuable insights into the system under test, they need to be executed in an environment where resilience wants to be enhanced. These experiments can be limited depending on the selected environment. Chaos engineering experimentation looks to create outages to gain insight. However, creating outages in production environments can be detrimental to the user and the business.

Another limitation of Chaos Engineering is the limitation of tools to certain infrastructures. Any experiment can be executed manually, but it must be scheduled and automized for continuous improvement and learning. This could lead to the need for custom tool creation depending on the infrastructure.

## 6.3 Experiments Improvements

The testing of this research can be improved by deploying to a specific cloud provider, or the final deployment strategy. This can provide new insights that are proper for cloud providers. The main goal is to have real-world resources so that performance cannot be skewed by testing factors.

For the selection of a cloud provider, the selection of the proper region also can be considered. This will affect the response time, which is the system performance indicator.

Automated testing can also be implemented in the real-world system, to keep the learning and enhancement continuous. Small outages that discover weak points in the system, can save potential bigger outages. These experiments should be randomly triggered, except for peak hours or important days.

Besides the response time, that is the system performance, the experiments can also consider resource utilization, scalability, and security. These are also important to the quality of well-designed applications.

## 6.4 Future Research

In future research, several points can be addressed for the enhancement of the system resilience.

The system can be deployed using a multi-cloud provider. This can greatly reduce the fault outages, enhancing the resilience depending on a cloud provider. However, this can also negatively affect the system performance since response times should increase, due to the services being in different networks.

The system also can enhance resilience against security issues. This can mean resource exhaustion, by networking attacks, brute force attacks, and more. The practices and strategies applied for security can also enhance resilience.

## Bibliography

[1] H. Su, Y. Wang, M. Li, and L. Jia, “Research and Design of Risk Assessment and Control System Microservice-Based Urban Rail Transit,” in *2021 7th Annual International Conference on Network and Information Systems for Computers (ICNISC)*, Jul. 2021, pp. 82–88. doi: 10.1109/ICNISC54316.2021.00024.

[2] M. Driss, D. Hasan, W. Boulila, and J. Ahmad, “Microservices in IoT Security: Current Solutions, Research Challenges, and Future Directions,” *Procedia Comput. Sci.*, vol. 192, pp. 2385–2395, Jan. 2021, doi: 10.1016/j.procs.2021.09.007.

[3] G. Zhang, K. Ren, J.-S. Ahn, and S. Ben-Romdhane, “GRIT: Consistent Distributed Transactions Across Polyglot Microservices with Multiple Databases,” in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, Macao, Macao: IEEE, Apr. 2019, pp. 2024–2027. doi: 10.1109/ICDE.2019.00230.

[4] M. K. Gokhroo, M. C. Govil, and E. S. Pilli, “Detecting and mitigating faults in cloud computing environment,” in *2017 3rd International Conference on Computational Intelligence & Communication Technology (CICT)*, Feb. 2017, pp. 1–9. doi: 10.1109/CIACT.2017.7977362.

[5] M. Villamizar *et al.*, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” in *2015 10th Computing Colombian Conference (10CCC)*, Sep. 2015, pp. 583–590. doi: 10.1109/ColumbianCC.2015.7333476.

[6] N. C. Mendonca, C. M. Aderaldo, J. Camara, and D. Garlan, “Model-Based Analysis of Microservice Resiliency Patterns,” *2020 IEEE Int. Conf. Softw. Archit. ICSA*, pp. 114–124, Mar. 2020, doi: 10.1109/ICSA47634.2020.00019.

[7] A. Sari and M. Akkaya, “Fault Tolerance Mechanisms in Distributed Systems,” *Int. J. Commun. Netw. Syst. Sci.*, vol. 8, no. 12, Art. no. 12, Dec. 2015, doi: 10.4236/ijcns.2015.812042.

[8] H. Adamu, B. Mohammed, A. B. Maina, A. Cullen, H. Ugail, and I. Awan, “An Approach to Failure Prediction in a Cloud Based Environment,” in *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, Aug. 2017, pp. 191–197. doi: 10.1109/FiCloud.2017.56.

[9] C. K. Rudrabhatla, “Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture,” *Int. J. Adv. Comput. Sci. Appl. Ijacsa*, vol. 9, no. 8, Art. no. 8, 49/01 2018, doi: 10.14569/IJACSA.2018.090804.

[10] H. Yu, X. Wang, C. Xing, and B. Xu, “A Microservice Resilience Deployment Mechanism Based on Diversity,” *Secur. Commun. Netw.*, vol. 2022, p. e7146716, Jun. 2022, doi: 10.1155/2022/7146716.

[11] D. Ilic and E. Troubitsyna, “Formal development of software for tolerating transient faults,” in *11th Pacific Rim International Symposium on Dependable Computing (PRDC’05)*, Dec. 2005, p. 8 pp.-. doi: 10.1109/PRDC.2005.34.

[12] C. Konstantinou, G. Stergiopoulos, M. Parvania, and P. Esteves-Verissimo, “Chaos Engineering for Enhanced Resilience of Cyber-Physical Systems,” in *2021 Resilience Week (RWS)*, Oct. 2021, pp. 1–10. doi: 10.1109/RWS52686.2021.9611797.

[13] J. Lu, J. Guo, Z. Jian, Y. Yang, and W. Tang, “Resilience Assessment and Its Enhancement in Tackling Adverse Impact of Ice Disasters for Power Transmission Systems,” *Energies*, vol. 11, no. 9, Art. no. 9, Sep. 2018, doi: 10.3390/en11092272.

[14] K. Dürr, R. Lichtenthaeler, and G. Wirtz, *An Evaluation of Saga Pattern Implementation Technologies*. 2021.

[15] J. Carnell, *Spring microservices in action*. in JAVA. Shelter Island: Manning, 2017.

[16] Michael T. Nygard, *Release It! Second Edition. Design and Deploy Production-Ready Software*. Andy Hunt, 2018.
[17] J. Turnbull, *The Docker Book: Containerization Is the New Virtualization*. James Turnbull, 2014.
[18] B. Burns, J. Beda, and K. Hightower, *Kubernetes: Up and Running*.
[19] G. M. Roy, *RabbitMQ in depth*. Shelter Island, NY: Manning Publications Co, 2018.
[20] D. Boike,David, *Learning NserviceBus*. in 2. 2015.
[21] C. Rosenthal and N. Jones, *Chaos Engineering: System Resiliency in Practice*. O'Reilly Media, Inc., 2020.
[22] VALLISADMIN, "STATE-OF-THE-ART OF MESSAGING FOR DISTRIBUTED COMPUTING SYSTEMS – International Journal Vallis Aurea." Accessed: Nov. 30, 2023. [Online]. Available: http://journal.vallisaurea.org/blog/2018/01/15/state-of-the-art-of-messaging-for-distributed-computing-systems/
[23] C. Cassé, P. Berthou, P. Owezarski, and S. Josset, "Using Distributed Tracing to Identify Inefficient Resources Composition in Cloud Applications," in *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*, Nov. 2021, pp. 40–47. doi: 10.1109/CloudNet53349.2021.9657140.
[24] A. Megargel, V. Shankararaman, and D. K. Walker, "Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example," in *Software Engineering in the Era of Cloud Computing*, M. Ramachandran and Z. Mahmood, Eds., in Computer Communications and Networks. , Cham: Springer International Publishing, 2020, pp. 85–108. doi: 10.1007/978-3-030-33624-0_4.
[25] P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study," *J. Syst. Softw.*, vol. 150, pp. 77–97, Apr. 2019, doi: 10.1016/j.jss.2019.01.001.
[26] J. Moura and D. Hutchison, "Fog computing systems: State of the art, research issues and future trends, with a focus on resilience," *J. Netw. Comput. Appl.*, vol. 169, p. 102784, Nov. 2020, doi: 10.1016/j.jnca.2020.102784.
[27] P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*. Morgan Kaufmann, 2009.
[28] C. Richardson, *Microservices Patterns: With examples in Java*. Manning, 2018. [Online]. Available: https://books.google.de/books?id=QTgzEAAAQBAJ
[29] N. Lynch, M. Merritt, W. Weihl, and A. Fekete, "A theory of atomic transactions," in *ICDT '88*, M. Gyssens, J. Paredaens, and D. Van Gucht, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1988, pp. 41–71. doi: 10.1007/3-540-50171-1_3.
[30] A. Zaid, "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation." OSF Preprints, Mar. 27, 2023. doi: 10.31219/osf.io/ksj5q.
[31] "On the Effectiveness of Tools to Support Infrastructure as Code: Model-Driven Versus Code-Centric | IEEE Journals & Magazine | IEEE Xplore." Accessed: Nov. 30, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/8959180
[32] G. Samaras, K. Britton, A. Citron, and C. Mohan, "Two-phase commit optimizations in a commercial distributed environment," *Distrib. Parallel Databases*, vol. 3, no. 4, pp. 325–360, Oct. 1995, doi: 10.1007/BF01299677.
[33] Y. Al-Houmaily, P. Chrysanthis, and S. P. Levitan, *An argument in favor of the presumed commit protocol*. 1997, p. 265. doi: 10.1109/ICDE.1997.581795.

[34] martinekuan, “Saga pattern - Azure Design Patterns.” Accessed: Nov. 18, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga
[35] “Two-Phased Locking, Releasing Locks & Deadlocks in Databases,” study.com. Accessed: Nov. 14, 2023. [Online]. Available: https://study.com/WEB-INF/views/jsp/redesign/academy/lesson/seoLessonPage.jsp
[36] P. T. Endo, M. Rodrigues, G. E. Gonçalves, J. Kelner, D. H. Sadok, and C. Curescu, “High availability in clouds: systematic review and research challenges,” *J. Cloud Comput.*, vol. 5, no. 1, p. 16, Oct. 2016, doi: 10.1186/s13677-016-0066-8.
[37] martinekuan, “Reliability patterns - Cloud Design Patterns.” Accessed: Nov. 10, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/well-architected/resiliency/reliability-patterns
[38] “Timeout resilience strategy | Polly.” Accessed: Nov. 19, 2023. [Online]. Available: https://www.pollydocs.org/strategies/timeout.html#diagrams
[39] “Message Headers • NServiceBus,” Particular Docs. Accessed: Nov. 19, 2023. [Online]. Available: https://docs.particular.net/nservicebus/messaging/headers
[40] “Correlation Identifier,” Enterprise Integration Patterns. Accessed: Nov. 19, 2023. [Online]. Available: https://www.enterpriseintegrationpatterns.com/patterns/messaging/CorrelationIdentifier.html
[41] Datadog, “What is Distributed Tracing? How it Works & Use Cases | Datadog,” What is Distributed Tracing? How it Works & Use Cases. Accessed: Nov. 19, 2023. [Online]. Available: https://www.datadoghq.com/knowledge-center/distributed-tracing/
[42] “Avoiding fallback in distributed systems,” Amazon Web Services, Inc. Accessed: Nov. 20, 2023. [Online]. Available: https://aws.amazon.com/builders-library/avoiding-fallback-in-distributed-systems/
[43] “Fallback resilience strategy | Polly.” Accessed: Nov. 20, 2023. [Online]. Available: https://www.pollydocs.org/strategies/fallback.html
[44] “Implement health check APIs for microservices - IBM Garage Practices.” Accessed: Nov. 20, 2023. [Online]. Available: https://www.ibm.com/garage/method/practices/manage/health-check-apis/
[45] jamesmontemagno, “Health monitoring - .NET.” Accessed: Nov. 20, 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/monitor-app-health
[46] prasha-microsoft, “Understand chaos engineering and resilience with Chaos Studio.” Accessed: Nov. 22, 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/chaos-studio/chaos-studio-chaos-engineering-overview
[47] C. K. Rudrabhatla, “A Quantitative Approach for Estimating the Scaling Thresholds and Step Policies in a Distributed Microservice Architecture,” *IEEE Access*, vol. 8, pp. 180246–180254, 2020, doi: 10.1109/ACCESS.2020.3028310.
[48] “Sensors | Free Full-Text | Moving Microgrid Hierarchical Control to an SDN-Based Kubernetes Cluster: A Framework for Reliable and Flexible Energy Distribution.” Accessed: Nov. 30, 2023. [Online]. Available: https://www.mdpi.com/1424-8220/23/7/3395
[49] A. Gupta, “Kubernetes: Overview,” Kubernetes. Accessed: Nov. 30, 2023. [Online]. Available: https://kubernetes.io/docs/concepts/overview/

[50] J. Mukaj, “Containerization: Revolutionizing Software Development and Deployment Through Microservices Architecture Using Docker and Kubernetes,” 2023. doi: 10.13140/RG.2.2.23804.51841.
[51] C. Zheng, Q. Zhuang, and F. Guo, “A Multi-Tenant Framework for Cloud Container Services.” arXiv, Mar. 24, 2021. doi: 10.48550/arXiv.2103.13333.
[52] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, “A mixed-method empirical study of Function-as-a-Service software development in industrial practice,” *J. Syst. Softw.*, vol. 149, pp. 340–359, Mar. 2019, doi: 10.1016/j.jss.2018.12.013.
[53] M. S. Rahaman, A. Islam, T. Cerny, and S. Hutton, “Static-Analysis-Based Solutions to Security Challenges in Cloud-Native Systems: Systematic Mapping Study,” *Sensors*, vol. 23, no. 4, Art. no. 4, Jan. 2023, doi: 10.3390/s23041755.
[54] “Helm.” Accessed: Dec. 07, 2023. [Online]. Available: https://helm.sh/
[55] “Deployments,” Kubernetes. Accessed: Dec. 07, 2023. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
[56] J. Adam, “The role of Helm in a Kubernetes architecture,” Jan. 2021. Accessed: Dec. 07, 2023. [Online]. Available: https://kruschecompany.com/helm-kubernetes/
[57] “Helm Architecture.” Accessed: Dec. 07, 2023. [Online]. Available: https://helm.sh/docs/topics/architecture/
[58] S. Böhm and G. Wirtz, “Cloud-Edge Orchestration for Smart Cities: A Review of Kubernetes-based Orchestration Architectures,” *EAI Endorsed Trans. Smart Cities*, vol. 6, no. 18, pp. e2–e2, May 2022, doi: 10.4108/eetsc.v6i18.1197.
[59] “HIGH AVAILABILITY AND DISASTER RECOVERY IN AZURE FOR KUBERNETES,” *Int. Res. J. Mod. Eng. Technol. Sci.*, Jan. 2023, doi: 10.56726/IRJMETS32765.
[60] D. Fahland *et al.*, “Declarative versus Imperative Process Modeling Languages: The Issue of Understandability,” in *Enterprise, Business-Process and Information Systems Modeling*, T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, and R. Ukor, Eds., in Lecture Notes in Business Information Processing. Berlin, Heidelberg: Springer, 2009, pp. 353–366. doi: 10.1007/978-3-642-01862-6_29.
[61] S. S. Gill, I. Chana, M. Singh, and R. Buyya, “RADAR: Self-configuring and self-healing in resource management for enhancing the quality of cloud services,” *Concurr. Comput. Pract. Exp.*, vol. 31, no. 1, p. e4834, 2019, doi: 10.1002/cpe.4834.
[62] M. Kennedy, S. Gonick, H. Meischke, J. Rios, and N. A. Errett, “Building Back Better: Local Health Department Engagement and Integration of Health Promotion into Hurricane Harvey Recovery Planning and Implementation,” *Int. J. Environ. Res. Public. Health*, vol. 16, no. 3, Art. no. 3, Jan. 2019, doi: 10.3390/ijerph16030299.
[63] R. B. Olshansky and L. A. Johnson, “The Evolution of the Federal Role in Supporting Community Recovery After U.S. Disasters,” *J. Am. Plann. Assoc.*, vol. 80, no. 4, pp. 293–304, Oct. 2014, doi: 10.1080/01944363.2014.967710.
[64] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, “Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration,” *Sensors*, vol. 20, no. 16, Art. no. 16, Jan. 2020, doi: 10.3390/s20164621.
[65] “Horizontal Pod Autoscaling,” Kubernetes. Accessed: Dec. 01, 2023. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[66] “Gateway API,” Kubernetes. Accessed: Dec. 01, 2023. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/gateway/
[67] “Volumes,” Kubernetes. Accessed: Dec. 02, 2023. [Online]. Available: https://kubernetes.io/docs/concepts/storage/volumes/

[68] “Persistent Volumes,” Kubernetes. Accessed: Dec. 02, 2023. [Online]. Available: https://kubernetes.io/docs/concepts/storage/persistent-volumes/
[69] “StatefulSets,” Kubernetes. Accessed: Dec. 02, 2023. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/
[70] “Run a Replicated Stateful Application,” Kubernetes. Accessed: Dec. 02, 2023. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/
[71] “Performing a Rolling Update,” Kubernetes. Accessed: Dec. 07, 2023. [Online]. Available: https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/
[72] T. Pohanka and V. Pechanec, “Evaluation of Replication Mechanisms on Selected Database Systems,” *ISPRS Int. J. Geo-Inf.*, vol. 9, no. 4, Art. no. 4, Apr. 2020, doi: 10.3390/ijgi9040249.
[73] B. Kemme and G. Alonso, “Database replication: a tale of research across communities,” *Proc. VLDB Endow.*, vol. 3, no. 1–2, pp. 5–12, Sep. 2010, doi: 10.14778/1920841.1920847.
[74] “Replication — MongoDB Manual.” Accessed: Dec. 02, 2023. [Online]. Available: https://www.mongodb.com/docs/manual/replication/
[75] W. Schultz, T. Avitabile, and A. Cabral, “Tunable consistency in MongoDB,” *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2071–2081, Aug. 2019, doi: 10.14778/3352063.3352125.
[76] G. Haughian, R. Osman, and W. J. Knottenbelt, “Benchmarking Replication in Cassandra and MongoDB NoSQL Datastores,” in *Database and Expert Systems Applications*, vol. 9828, S. Hartmann and H. Ma, Eds., in Lecture Notes in Computer Science, vol. 9828. , Cham: Springer International Publishing, 2016, pp. 152–166. doi: 10.1007/978-3-319-44406-2_12.
[77] “What Is Replication In MongoDB?,” MongoDB. Accessed: Dec. 02, 2023. [Online]. Available: https://www.mongodb.com/basics/replication
[78] “PRINCIPLES OF CHAOS ENGINEERING - Principles of chaos engineering.” Accessed: Dec. 08, 2023. [Online]. Available: https://principlesofchaos.org/
[79] I. Karabey Aksakalli, T. Çelik, A. B. Can, and B. Tekinerdoğan, “Deployment and communication patterns in microservice architectures: A systematic literature review,” *J. Syst. Softw.*, vol. 180, p. 111014, Oct. 2021, doi: 10.1016/j.jss.2021.111014.
[80] R. A. Kass, “Tests and Experiments: Similarities and Differences,” presented at the Live-Virtual-Constructive Conference, El Paso, Texas: The ITEA Journal of Test and Evaluation, Dec. 2009.
[81] R. G. Sargent, “Verification and validation of simulation models,” in *2007 Winter Simulation Conference*, Dec. 2007, pp. 124–137. doi: 10.1109/WSC.2007.4419595.
[82] “PingCAP, the company behind TiDB,” PingCAP. Accessed: Dec. 11, 2023. [Online]. Available: https://www.pingcap.com/
[83] “Chaos Mesh Overview | Chaos Mesh.” Accessed: Dec. 11, 2023. [Online]. Available: https://chaos-mesh.org/docs/
[84] “Custom Resources,” Kubernetes. Accessed: Dec. 11, 2023. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/
[85] “Simulate Pod Faults | Chaos Mesh.” Accessed: Dec. 12, 2023. [Online]. Available: https://chaos-mesh.org/docs/simulate-pod-chaos-on-kubernetes/
[86] “Simulate Network Faults | Chaos Mesh.” Accessed: Dec. 12, 2023. [Online]. Available: https://chaos-mesh.org/docs/simulate-network-chaos-on-kubernetes/

[87] “Simulate Stress Scenarios | Chaos Mesh.” Accessed: Dec. 12, 2023. [Online]. Available: https://chaos-mesh.org/docs/simulate-heavy-stress-on-kubernetes/
[88] G. Pavlou and I. Psaras, “The troubled journey of QoS: From ATM to content networking, edge-computing and distributed internet governance,” *Comput. Commun.*, vol. 131, pp. 8–12, Oct. 2018, doi: 10.1016/j.comcom.2018.07.006.
[89] S. Sharwood and A. Editor, “AWS power failure killed some hardware and instances.” Accessed: Dec. 12, 2023. [Online]. Available: https://www.theregister.com/2021/12/22/aws_outage/
[90] X. Li, H. Wang, S. Yi, and L. Zhai, “Cost-efficient disaster backup for multiple data centers using capacity-constrained multicast,” *Concurr. Comput. Pract. Exp.*, vol. 31, no. 17, p. e5266, 2019, doi: 10.1002/cpe.5266.
[91] A. Ćatović, N. Buzađija, and S. Lemes, “Microservice development using RabbitMQ message broker,” *Sci. Eng. Technol.*, vol. 2, no. 1, Art. no. 1, Apr. 2022, doi: 10.54327/set2022/v2.i1.19.
[92] S. Weerasinghe and I. Perera, “Optimized Strategy for Inter-Service Communication in Microservices,” *Int. J. Adv. Comput. Sci. Appl. IJACSA*, vol. 14, no. 2, Art. no. 2, Dec. 2023, doi: 10.14569/IJACSA.2023.0140233.
[93] “AMQP is the Internet Protocol for Business Messaging | AMQP.” Accessed: Dec. 15, 2023. [Online]. Available: https://www.amqp.org/about/what
[94] “RabbitMQ tutorial - Work Queues — RabbitMQ.” Accessed: Dec. 14, 2023. [Online]. Available: https://www.rabbitmq.com/tutorials/tutorial-two-dotnet.html
[95] “NServiceBus Step-by-step: Retrying errors,” Particular Docs. Accessed: Dec. 20, 2023. [Online]. Available: https://docs.particular.net/tutorials/nservicebus-step-by-step/5-retrying-errors/
[96] “Persistence • NServiceBus,” Particular Docs. Accessed: Dec. 20, 2023. [Online]. Available: https://docs.particular.net/persistence/
[97] “Saga Timeouts • NServiceBus,” Particular Docs. Accessed: Dec. 20, 2023. [Online]. Available: https://docs.particular.net/nservicebus/sagas/timeouts
[98] “What is .NET? An open-source developer platform.,” Microsoft. Accessed: Dec. 20, 2023. [Online]. Available: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet
[99]. “NET customers showcase | See what devs are building,” Microsoft. Accessed: Dec. 20, 2023. [Online]. Available: https://dotnet.microsoft.com/en-us/platform/customers
[100] G. Blinowski, A. Ojdowska, and A. Przybyłek, “Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation,” *IEEE Access*, vol. 10, pp. 20357–20374, 2022, doi: 10.1109/ACCESS.2022.3152803.
[101] C. H. Gammelgaard, *Microservices in .NET, Second Edition*. Simon and Schuster, 2021.
[102] Y. Graham and Q. Liu, “Achieving Accurate Conclusions in Evaluation of Automatic Machine Translation Metrics,” in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, K. Knight, A. Nenkova, and O. Rambow, Eds., San Diego, California: Association for Computational Linguistics, Jun. 2016, pp. 1–10. doi: 10.18653/v1/N16-1001.
[103] A. Parker, D. Spoonhower, J. Mace, and R. Isaacs, *Distributed Tracing in Practice*, vol. 1. 2020.

[104] S. Souders, “Velocity and the Bottom Line - O’Reilly Radar.” Accessed: Dec. 30, 2023. [Online]. Available: http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html
[105] “Observability Primer,” OpenTelemetry. Accessed: Dec. 30, 2023. [Online]. Available: https://opentelemetry.io/docs/concepts/observability-primer/
[106] V. Anand, M. Stolet, T. Davidson, I. Beschastnikh, T. Munzner, and J. Mace, “Aggregate-Driven Trace Visualizations for Performance Debugging,” arXiv.org. Accessed: Dec. 30, 2023. [Online]. Available: https://arxiv.org/abs/2010.13681v1
[107] A. Klenik and A. Pataricza, “Adding Semantics to Measurements: Ontology-Guided, Systematic Performance Analysis,” *Acta Cybern.*, vol. 26, no. 2, Art. no. 2, 2023, doi: 10.14232/actacyb.295182.
[108] “Introduction,” Jaeger: open source, distributed tracing platform. Accessed: Dec. 30, 2023. [Online]. Available: https://www.jaegertracing.io/docs/1.52/
[109] karelz, “HttpClient.Timeout Property (System.Net.Http).” Accessed: Dec. 31, 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.net.http.httpclient.timeout?view=net-8.0
[110] I. B. Muller *et al.*, “Computational comparison of common event-based differential splicing tools: practical considerations for laboratory researchers,” *BMC Bioinformatics*, vol. 22, no. 1, p. 347, Jun. 2021, doi: 10.1186/s12859-021-04263-9.
[111] “Automatically Discovering, Reporting and Reproducing Android Application Crashes | IEEE Conference Publication | IEEE Xplore.” Accessed: Jan. 03, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/7515457
[112] “Transient Fault Detection in Networked Control Systems - Xiong-Feng Huang, Chun-Jie Zhou, Shuang Huang, Kai-Xin Huang, Xuan Li, 2014.” Accessed: Jan. 05, 2024. [Online]. Available: https://journals.sagepub.com/doi/10.1155/2014/346269

## Appendix

```
##api microservice
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orchestratorapi-deployment
  labels:
    app: orchestratorapi
spec:
  replicas: 1
  selector:
    matchLabels:
      app: orchestratorapi
  template:
    metadata:
      labels:
        app: orchestratorapi
    spec:
      containers:
        - name: orchestratorapi
          image: mahamtr/api_srv:latest
          env:
            - name: OTEL_EXPORTER_JAEGER_AGENT_HOST
              value: simplest-agent
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: orchestratorapi-service
spec:
  type: ClusterIP
  ports:
    - port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: orchestratorapi
## kubectl port-forward {{podname}} 8080:8080

## payment-service
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment-deployment
  labels:
    app: payment
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: payment
  template:
    metadata:
      labels:
        app: payment
    spec:
      containers:
        - name: payment
          image: mahamtr/payments_srv:latest
          env:
            - name: OTEL_EXPORTER_JAEGER_AGENT_HOST
              value: simplest-agent
          imagePullPolicy: Always
---
## order-service
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-deployment
  labels:
    app: order
spec:
  replicas: 1
  selector:
    matchLabels:
      app: order
  template:
    metadata:
      labels:
        app: order
    spec:
      containers:
        - name: order
          image: mahamtr/orders_srv:latest
          env:
            - name: OTEL_EXPORTER_JAEGER_AGENT_HOST
              value: simplest-agent
          imagePullPolicy: Always
---
## notification-service
apiVersion: apps/v1
kind: Deployment
metadata:
  name: notification-deployment
  labels:
    app: notification
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: notification
  template:
    metadata:
      labels:
        app: notification
    spec:
      containers:
        - name: notification
          image: mahamtr/notifications_srv:latest
          env:
            - name: OTEL_EXPORTER_JAEGER_AGENT_HOST
              value: simplest-agent
          imagePullPolicy: Always
---
## inventory-service
apiVersion: apps/v1
kind: Deployment
metadata:
  name: inventory-deployment
  labels:
    app: inventory
spec:
  replicas: 1
  selector:
    matchLabels:
      app: inventory
  template:
    metadata:
      labels:
        app: inventory
    spec:
      containers:
        - name: inventory
          image: mahamtr/inventory_srv:latest
          env:
            - name: OTEL_EXPORTER_JAEGER_AGENT_HOST
              value: simplest-agent
          imagePullPolicy: Always
---
```

Listing 1 Kubernetes Services Deployment Declaration

```
using SharedMessages.Enums;
using SharedMessages.Events;
using SharedMessages.ResponseEvents;
using SharedMessages.Timeout;
```

```
namespace Api.Gateway.Sagas;

public class PlaceOrderSaga(IConfiguration configuration) :
Saga<PlaceOrderSagaData>, IAmStartedByMessages<StartOrder>,
    IHandleMessages<InventoryUpdated>,
IHandleMessages<PaymentSucceed>,
IHandleMessages<OrdersCreated>,
    IHandleMessages<EndOrderSuccess>,
IHandleMessages<RejectOrder>,
IHandleMessages<RollbackSuccess>,IHandleTimeouts<CreateOrde
rTimeout>
{
    private SendOptions Options { get; set; } = new();

    protected override void
ConfigureHowToFindSaga(SagaPropertyMapper<PlaceOrderSagaDat
a> mapper)
    {
        mapper.MapSaga(sagaData => sagaData.OrderId)
            .ToMessage<StartOrder>(message =>
message.OrderId)
            .ToMessage<InventoryUpdated>(message =>
message.OrderId)
            .ToMessage<PaymentSucceed>(message =>
message.OrderId)
            .ToMessage<OrdersCreated>(message =>
message.OrderId)
            .ToMessage<EndOrderSuccess>(message =>
message.OrderId)
            .ToMessage<RejectOrder>(message =>
message.OrderId)
            .ToMessage<RollbackSuccess>(message =>
message.OrderId)
            .ToMessage<CreateOrderTimeout>(message =>
message.OrderId);
    }

    public async Task Handle(StartOrder message,
IMessageHandlerContext context)
    {
        Data.OrderId = message.OrderId;
        Data.InventoryId = message.InventoryId;
        Data.Quantity = message.Quantity;

        await RequestTimeout(context,
TimeSpan.FromMinutes(1), new CreateOrderTimeout { OrderId =
Data.OrderId });
```

```
Options.SetDestination(configuration.GetSection("InventoryE
ndpointName").Value + "-MAAI");
        await context.Send(
            new UpdateInventory { OrderId =
message.OrderId, Quantity = Data.Quantity, InventoryId =
Data.InventoryId },
            Options);
    }

    public async Task Handle(InventoryUpdated message,
IMessageHandlerContext context)
    {
        Data.MessageData = message.MessageData;


Options.SetDestination(configuration.GetSection("PaymentsEn
dpointName").Value + "-MAAI");
        await context.Send(new StartPayment { OrderId =
Data.OrderId }, Options);
    }

    public async Task Handle(PaymentSucceed message,
IMessageHandlerContext context)
    {
        Data.PaymentId = message.PaymentOrderId;
        Data.MessageData = message.MessageData;


Options.SetDestination(configuration.GetSection("OrdersEndp
ointName").Value + "-MAAI");
        await context.Send(new CreateOrders { OrderId =
Data.OrderId }, Options);
    }

    public async Task Handle(OrdersCreated message,
IMessageHandlerContext context)
    {
        Data.ShippingOrderId = message.ShipmentOrderId;
        Data.PurchaseOrderId = message.PurchaseOrderId;
        Data.MessageData = message.MessageData;


Options.SetDestination(configuration.GetSection("Notificati
onsEndpointName").Value + "-MAAI");
        await context.Send(new NotifyCustomer { OrderId =
Data.OrderId }, Options);
    }

    public async Task Handle(EndOrderSuccess message,
IMessageHandlerContext context)
```

```
    {
        Data.HasResolved = true;
        Data.MessageData = message.MessageData;

        await ReplyToOriginator(context,
            new EndOrderSuccess { OrderId = Data.OrderId,
MessageData = "Successful Request" });
        MarkAsComplete();
    }

    public async Task Timeout(CreateOrderTimeout message,
IMessageHandlerContext context)
    {
        if (Data.HasResolved) return;
        await ReplyToOriginator(context,
            new EndOrderSuccess { OrderId = Data.OrderId,
MessageData = "Timeout has been reached" });
        MarkAsComplete();
    }

    public async Task Handle(RejectOrder message,
IMessageHandlerContext context)
    {

Options.SetDestination(configuration.GetSection("InventoryE
ndpointName").Value + "-MAAI");
        switch (message.Failure)
        {
            case CreateOrderFailures.PaymentFailure:
                await context.Send(new
RollbackInventory{OrderId = Data.OrderId, InventoryId =
Data.InventoryId,Quantity = Data.Quantity},Options);
                break;
            case CreateOrderFailures.OrdersCreationFailure:
                await context.Send(new RollbackInventory
{OrderId = Data.OrderId, InventoryId =
Data.InventoryId,Quantity = Data.Quantity},Options);

Options.SetDestination(configuration.GetSection("PaymentsEn
dpointName").Value + "-MAAI");
                await context.Send(new RollbackPayment
{OrderId = Data.OrderId, PaymentOrderId = Data.PaymentId});
                break;
            default:
                await ReplyToOriginator(context,
                    new EndOrderSuccess
                    {
                        OrderId = Data.OrderId,
                        MessageData = "Order has been
rejected with no rollback requeriment."
```

```
                    });
                MarkAsComplete();
                break;
        }
    }

    public async Task Handle(RollbackSuccess message,
IMessageHandlerContext context)
    {
        var step = message.Step;
        if (step == RollbackTypes.InventoryRollback &&
Data.PaymentId == null)
        {
            await ReplyToOriginator(context,
                new EndOrderSuccess
                {
                    OrderId = Data.OrderId,
                    MessageData = "Order has been rejected
and successfully rollback to inventory."
                });
            MarkAsComplete();
        }

        if (step == RollbackTypes.InventoryRollback)
        {
            Data.InventoryHasBeenRollback = true;
        }

        if (step == RollbackTypes.PaymentRollback)
        {
            Data.PaymentHasBeenRollback = true;
        }

        if (step == RollbackTypes.InventoryRollback &&
Data.PaymentHasBeenRollback)
        {
            await ReplyToOriginator(context,
                new EndOrderSuccess
                {
                    OrderId = Data.OrderId,
                    MessageData = "Order has been rejected
and successfully rollback to inventory and payment."
                });
            MarkAsComplete();
        }

        if (step == RollbackTypes.PaymentRollback &&
Data.InventoryHasBeenRollback)
        {
            await ReplyToOriginator(context,
```

```
                new EndOrderSuccess
                {
                    OrderId = Data.OrderId,
                    MessageData = "Order has been rejected
and successfully rollback to inventory and payment."
                });
            MarkAsComplete();
        }
    }
}

public class PlaceOrderSagaData : ContainSagaData
{
    public Guid OrderId { get; set; }
    public string? InventoryId { get; set; }
    public int Quantity { get; set; }
    public string? ShippingOrderId { get; set; }
    public string? PaymentId { get; set; }
    public string MessageData { get; set; } = string.Empty;
    public string? PurchaseOrderId { get; set; }
    public bool HasResolved { get; set; }
    public bool PaymentHasBeenRollback { get; set; }
    public bool InventoryHasBeenRollback { get; set; }
}
```

Listing 2 Saga and Data Model Declaration