From monolith to microservices: building scalable applications with Kubernetes, CI/CD, and chaos engineering

Bachelor's Thesis

Degree Programme in Business Information Technology

Spring 2025

Ahmed Mahgoub

HAMK
Häme University
of Applied Sciences

Abstract

DP	Business Information Technology
Author	Ahmed Mahgoub Year 2025
Subject	From monolith to microservices: building scalable applications with Kubernetes, CI/CD, and chaos engineering
Supervisors	Tero Keso

This thesis explores the development and resilience evaluation of an application designed using a microservices architecture and deployed on a local Kubernetes cluster. The primary objective was to assess the application's ability to withstand various failure scenarios by applying chaos engineering principles, using the Chaos Mesh platform. The research involved the systematic injection of faults – including pod failures, network disruptions and resource constraints – to uncover vulnerabilities and evaluate the system’s fault tolerance.

The experiments revealed critical issues, such as concurrent user account creation failures under network latency and exposed the differing resilience level of different technology stacks within the application. Additionally, a questionnaire was conducted to assess awareness and perceptions of chaos engineering within the IT community. The responses indicated growing interest, despite limited widespread familiarity.

The findings of this work demonstrate the practical value of chaos engineering in uncovering systemic weaknesses and underscore the importance of sound, proactive resilience strategies in modern microservices-based systems. This work contributes to a deeper understanding of how to build and test resilient cloud-native applications and highlights the potential for broader adoption of chaos engineering practices within the IT industry. It also provides valuable insights for universities and technical education institutions that may wish to incorporate resilience engineering concepts into their software engineering and DevOps curricula.

Keywords	DevOps, Microservices, Chaos engineering, Cloud-Native, Resilient application design
Pages	82 pages and appendices 20 pages

Table of Contents

Figures

Tables

Appendices

Appendix 1. Data management plan

Appendix 2. Vagrantfile Virtual Machine Script

Appendix 3. Example Kubernetes Deployment File

Appendix 4. Setup Automation Bash Script

Appendix 5. Stress Reliability Tests

Appendix 6. Chaos Engineering Questionnaire Responses

1 Introduction

In the current IT landscape, businesses are pressured to deliver and maintain scalable, resilient, and robust software applications. Traditional monolithic development architectures, which have served their purpose in the past, often face challenges in meeting requirements such as handling growing numbers of users, more complex functionalities, and the need for frequent updates. This led to a remarkable shift to microservices, which offer a more modular, flexible, and agile approach to fulfilling these requirements (Hochstetler, 2024).

To effectively realise the benefits of microservices, a robust ecosystem of technologies and practices is essential. Kubernetes has emerged as the standard software for orchestrating containers, providing necessary infrastructure for deploying, scaling and managing microservices (Gaur, 2021). Complementing Kubernetes, Continuous Integration and Continuous Delivery (CI/CD) pipelines automate the software development lifecycle, enabling faster and more reliable releases (Sahoo, 2025). Additionally, to ensure the resilience and stability of these systems, Chaos Engineering has gained prominence as a proactive approach to identify and address potential weaknesses and points of failure (OpenText, n.d.).

The practical aspects of building a modern, microservices-based application, leveraging Kubernetes for orchestration, CI/CD for streamlined deployments, and Chaos Engineering for failure point identification, are explored in this thesis. By examining the interaction between these technologies in a realistic scenario, we can gain valuable insights into building applications that would meet today's business needs. To achieve this goal, the following research questions are addressed:

- What are the limitations of traditional monolithic architectures in today’s IT world, and how do microservices address these challenges?
- In what ways does Kubernetes facilitate the deployment, scaling, and management of microservices compared to other methods?
- What is the role of Chaos Engineering in proactively identifying system vulnerabilities?
- What are the business benefits and risk management impacts of moving from a monolithic to a microservices-based architecture with integrated CI/CD and chaos engineering practices?

2 Methodology

This chapter details the methodology used in this research to explore the integration of Chaos Engineering within a microservices architecture and its impact on application resilience. The approach undertaken was primarily exploratory and experiential, involving the development of a microservices-based application as a practical learning platform. This practical approach enabled a more comprehensive understanding of microservices development, as well as the principles and practical application of CI/CD and Chaos Engineering in a realistic context.

2.1 Research approach and data collection methods

The research followed an applied approach, in which the creation and experimentation with a microservices application served as the primary method of investigation. This approach is consistent with the exploratory nature of the research, enabling direct interaction with real-world scenarios and observation of system behaviour under different conditions. Several different data collection methods were used to assess the results of the practice and to answer the research questions:

To gain insight into the microservice's internal state and behaviour during normal operation and under stress, specific application-level logs were closely monitored. The logs generated by the microservices application were a primary source of data, providing insights into application behaviour and errors, as well as responses to injected failures. These logs were then aggregated and analysed using Loki. Loki is a log aggregation system designed for efficient and scalable log management across multiple services.

Prometheus was used to collect time-series data on system performance and resource utilisation. Grafana provided visual dashboards to monitor key metrics such as CPU usage, memory consumption, network latency and error rates, both during normal operation and during the chaos experiments.

Practical user testing was carried out, particularly during the chaos experiments, to observe the behaviour and responsiveness of the application from an end-user perspective. This approach enabled the identification of any functional disruptions or performance degradations that might not be immediately apparent from system logs and metrics alone.

A questionnaire was distributed to gather feedback and insights into what students and industry professionals think about chaos engineering.

2.2 Functional testing and monitoring

Functional testing played a crucial role in ensuring the basic operational integrity of the developed microservices application. Prior to the implementation of Chaos Engineering experiments, functional testing was conducted to verify that the core functions of the application worked as expected under normal conditions.

In addition, functional tests were carried out concurrently with some of the chaos experiments to assess whether the injected faults caused any critical functional failures or unexpected behaviour from a user perspective. This approach helped to distinguish between performance degradation and complete functional failure. Continuous monitoring was a critical aspect of this research, particularly during the execution of chaos experiments. The following tools were used to monitor the system: Loki: For aggregating and querying application logs, providing detailed insights into the behaviour of individual microservices. Grafana: For visualizing key performance indicators and metrics collected by Prometheus, offering a real-time overview of the system's health and performance. Custom dashboards were created to track relevant metrics during experiments. Prometheus: The primary metrics collection system, Prometheus gathered data on resource utilization, network statistics, and other relevant system-level metrics. K9s: Terminal-based UI for interacting with Kubernetes clusters (k3s in this case), providing a real-time view of the health and status of pods, deployments, and other Kubernetes resources.

2.3 Ethical considerations and AI usage

The data collected from the questionnaire was fully anonymised to ensure the privacy and confidentiality of the participants' responses.

Furthermore, all experiments were conducted within a local development environment (k3s) specifically set up for this research. No production systems or real user data were involved, thereby mitigating any potential risks associated with injecting failures into live environments.

Large Language Models (LLMs) such as ChatGPT, Gemini and GitHub Copilot were utilised as supportive tools throughout the research process. These tools were employed for: Coding Assistance: This includes the generation of code snippets, providing suggestions for implementation, and assistance in troubleshooting development issues. Ideation for Chaos Experiments: The brainstorming process will include potential failure

scenarios and exploration of different types of chaos experiments that could be relevant to the microservices application's technology stack. Text Refinement for Documentation: Assisting in the writing, editing process, with a focus on ensuring clarity, coherence, and the appropriate tone for the thesis documentation.

2.4 Chaos experiments and evaluation criteria

The design of the chaos experiments was based on understanding the application, in terms of the technology stack and topology/architecture. This helped identify potential failure scenarios and helped with the design of experiments to simulate these conditions. The following experiments were carried out. The first experiment focused on resource exhaustion, where a simulated Out-Of-Memory (OOM) scenario and high CPU utilisation was introduced to microservice pods. The second experiment involved network disruption, by temporarily adding network delays, jitter and packet loss into microservices communication. The third experiment was pod termination, in which individual pods were forcibly killed to observe how the system responded to the failure and how it recovered from such failures. During these experiments, the behaviour of the system was monitored using the tools mentioned above (Loki, Grafana, Prometheus and K9s). The success and efficacy of the practical implementation and outcomes of the chaos experiments were evaluated through analysis of the findings and conclusions drawn from the experiments. The process involved a thorough inspection of the logs and metrics to identify weaknesses in the application's architecture, understand the impact of various failure scenarios, and uncover potential bugs or unexpected behaviour that may not have been evident during the standard development and functional testing phases. The application's ability to maintain functionality and recover from failures served as a key indicator of its resilience, as measured by monitoring and manual testing. This methodology enabled a pragmatic exploration of integrating chaos engineering into a microservices architecture, where the primary evaluation involved obtaining and examining system reports, logs and metrics to gain valuable insights and learn about the resilience characteristics of the developed application and its fault mechanisms.

3 Microservices

This chapter presents microservices, a key architectural style for building scalable and resilient applications. The chapter will define the core concepts of microservices, compare them to traditional monolithic architectures, and explore their benefits and challenges. The subsequent discussion on Kubernetes, CI/CD, and Chaos Engineering will build on this understanding.

3.1 Defining microservices

Before defining microservices, we must explain the term “cloud-native” first. Cloud-native is an approach that refers to how an application or a piece of software is built, deployed and managed in cloud environments. Cloud-native applications are designed to be flexible, scalable and resilient applications. They are designed in such a manner that they can be quickly updated, to meet companies and customers ever-changing demands (AWS, n.d.-e).

Microservices are an approach that refers to how software is organised. Applications are broken down into smaller functional units that communicate through various well-defined protocols. These microservices are owned, managed and maintained by self-contained software development teams. Microservices also allow for faster and more scalable applications (AWS, n.d.-c).

Microservices are an evolution of another architecture used in the late 1990s. Service-oriented architecture relied on exposing applications using standard network protocols like ‘Simple Object Access Protocol’ (SOAP). Today, microservices can use a multitude of ways to communicate both internally and externally, i.e. over the internet (Red Hat, 2023a; Red Hat, 2023b).

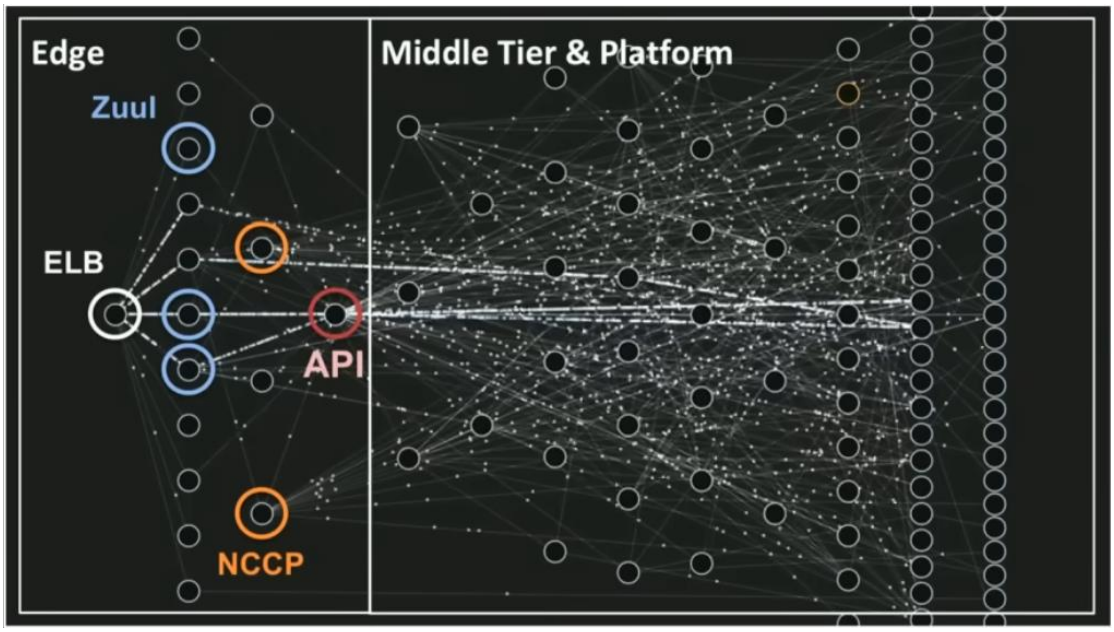
While SOA architecture was designed to integrate contrasting services over a network, microservices take this concept and break down applications further into even smaller, specialised, autonomous services. These services can communicate using a variety of methods — both synchronous (e.g., RESTful APIs) and asynchronous (e.g., message queues, event-driven) — and are often deployed using modern containerisation and orchestration technologies like Docker or Kubernetes (Red Hat, 2023a; GeeksforGeeks & Rawat, 2025).

3.2 The rise of microservices

Before microservices gained popularity in the 2010s, monoliths were the standard when developing software solutions. However, monoliths have their limitations when it comes to scalability, maintainability and speed of development. Microservices came as an answer to these limitations. Unlike monoliths, microservices are smaller units that operate independently. They are designed to perform specific business functions (Rao, 2023).

Early pioneers of microservices in the tech industry were companies like Netflix and Amazon. They were among the first companies to adopt this paradigm shift, having identified the potential of microservices, which helped them create their success stories. Figure 1 presents Netflix's system architecture in 2017, showing the flow from the edge to internal services. Systems that could be developed with minimal downtime, teams which operated independently as knowledge and specialisation silos, scaling seamlessly to meet the ever-increasing demand from customers, all that enabled businesses to quickly capitalise on possible opportunities. Netflix, for example, has successfully broken down its application into microservices and today they are running over 1000+ of them, each handling a specific part of the whole application, all while being scalable enough to meet fluctuations in demand (GeeksforGeeks & Venugopal, 2024).

Figure 1. Netflix's microservice architecture (InfoQ, 2017).



3.3 Core principles of microservices

As highlighted in the previous chapter with the examples of early pioneers like Netflix and Amazon, microservices changed the way large-scale applications are built and deployed. The adoption of microservices represents thus a significant departure from traditional monolithic architectures. Offering a range of advantages in terms of speed of development and fault tolerance helped microservices gain edge on their monolith counterparts. This chapter will provide an overview of the essential and core principles that make such a significant paradigm shift possible.

3.3.1 Autonomy and independence

Each microservice within the application should be self-contained, meaning that it must operate independently of other microservices. Each service should have its own resource pool, such as a separate database and its own business logic without relying on other services in case of unexpected scenarios (GeeksforGeeks & achal11sp, 2024).

It also allows for independent development, deployment, and release cycles. Each microservice is isolated from other ones, which means that teams can develop, test, and release updates or changes independently, eliminating the need to rely on or thoroughly coordinate with other teams (M, 2024).

Having isolated and autonomous services enables having technological diversity. This autonomy allows for different microservices to be built using the programming language, framework or tool that fits best for the job. This gives teams the freedom to be flexible and not be constrained by one single technology for the entire application (GeeksforGeeks & achal11sp, 2024).

Scalability is also one of the many benefits of microservices being independent. Each service can be scaled independently based on its resource and demand. For example, a user authentication service might have different workload patterns than a file processing service. With autonomy and independence from other microservices, it is possible to scale only the heavily used services without having to scale the entire system, which means a more efficient resources and cost optimization (M, 2024).

3.3.2 Failure isolation

In a monolithic architecture, all components of an application are tightly coupled, which means that if one part is to encounter a fatal error or becomes unavailable, it could potentially bring down the whole system. This subjects the availability of the application to the weakest link in the system.

Developers should always be aware and prepared for encountering unexpected scenarios, like failures in their applications. For example, if a microservice would fail due to an unexpected error, bug or maybe if the service is overloaded, the autonomy ensures that it does not directly impact the functionality of other microservices or the whole application. The failure is contained and limited to that specific service, preventing a stream of failures that can bring down the entire system (GeeksforGeeks & achal11sp, 2024).

By isolating failures, the application would often be able to gracefully degrade some of its functionality, instead of crashing in its entirety (GeeksforGeeks & dibyabrata1234, 2024). For example, if a service is responsible for insights in a banking application was to unexpectedly crash or go offline, the core functionality of the banking application would still be available. The user might simply not get insights into their financial situation while checking their balance or making payments would still be possible.

In the event of a failure in an isolated microservice, it is often easier and sometimes even faster to diagnose and recover from the issue. The team responsible can then focus their efforts on the specific failing part without having to worry about the stability of the entire system. That approach also reduces time spent on identifying the source of the failure, allowing teams to focus on recovering from the issue at hand in a promptly manner (GeeksforGeeks & Rawat, 2025).

Taking a second look at Figure 1, which shows the microservice architecture Netflix used in 2017, we can see how failure isolation might apply. For example, if a service within the “Middle Tier & Platform” experiences a failure, the “Edge” services like “Zuul” or “ELB” can be designed to gracefully handle such a failure, what maybe means redirecting traffic to healthy instances of the same service or by degrading the functional service to the user.

Several architectural patterns and practices help in achieving failure isolation in microservices, some of which are (GeeksforGeeks & gpancomputer, 2024): By creating specialised and independent business oriented microservices. Using asynchronous patterns like message queues or event streams can enhance failure isolation. Ideally, each microservice would have its own dedicated resources, such as database connections,

thread pools, and memory allocation. Circuit breaker patterns, Much like a circuit breaker in a home protects against electrical overload. This pattern prevents cascading failures in a microservices architecture by stopping requests to an unhealthy or failing service.

3.3.3 Specialised services

The principle of specialised services is central to the microservices architecture. It emphasises that each individual service should be built around solving a specific, well-defined business problem. For example, in an e-commerce platform, you might have specific services handling order management, product catalogues, user profiles, and processing payments. (GeeksforGeeks & Rawat, 2025).

Having specialised services follows the ‘single responsibility principle’ (SRP), a principle originally from object-oriented design. It suggests that each object or service should have “one reason to change”, which leads to more maintainable and cohesive services (Oso, n.d.). Following the SRP means modifications and updates are less likely to introduce unintended side effects in other parts of the application.

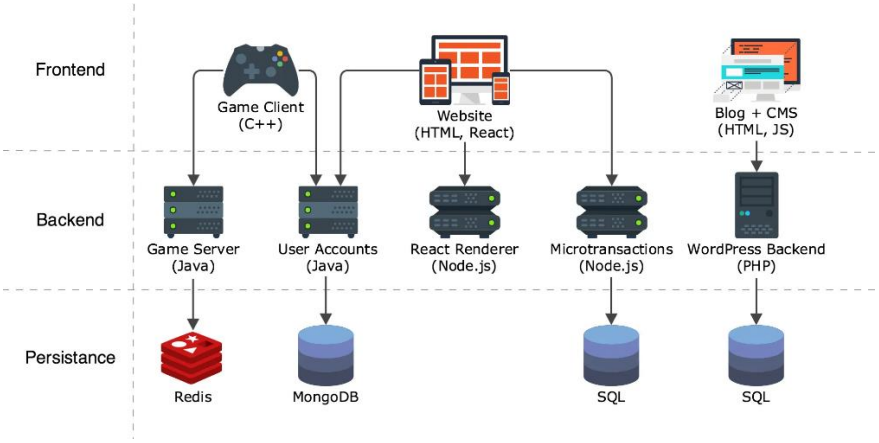
Designing specialised services makes it easier for teams to work on different parts of the application without impacting others, giving them an independent development life cycle.

3.3.4 Decentralised governance

The principle of decentralised governance means that there is no single, centralised standard or stack that all microservices need to adhere to. Teams have all freedom to choose the best tools, databases, programming languages and frameworks that are most suitable for their business problem. This gives teams more space to innovate and makes the adoption of new technologies possible without impacting the whole system (Peck, 2018a).

To illustrate decentralised governance in practice, an article written by user Nathan Peck on Medium describes a hypothetical game development studio. The studio has different engineering teams responsible for different service such as developing the game client, game server, user accounts and microtransactions payments. These teams are given the autonomy to select the most suitable tech stack for their respective service, for instance, the team responsible for developing the game client would use C++ to ensure performance, Java would be used for the game server, and NodeJS for processing the microtransactions (Peck, 2018a). Figure 2 visualises the situation described by Peck.

Figure 2. Software architecture for a hypothetical game studio (Peck, 2018a).



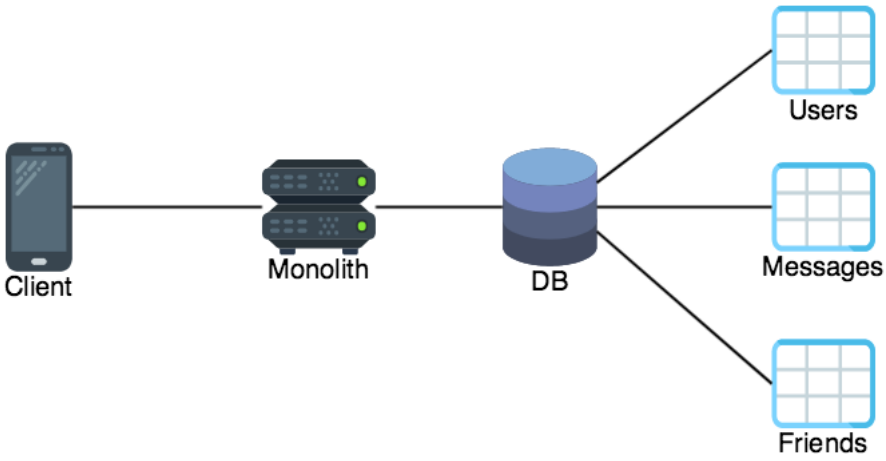
This demonstrates a decentralised approach where tech stack choices are driven by the expertise of each team and the specific demands at hands, rather than a centralised approach governing the development process.

3.3.5 Decentralised data management

Contrary to monolithic architectures that rely on a single, shared database, each microservice typically manages its own database or data store. This ensures that each service has total control over its data model and data access layer (DAL), which allows for data autonomy and prevents tight coupling at the data layer. This approach also enables using different database technologies specifically to meet the needs and requirements of each service (e.g., a graph database like GraphQL, a document store like MongoDB) (Peck, 2018b).

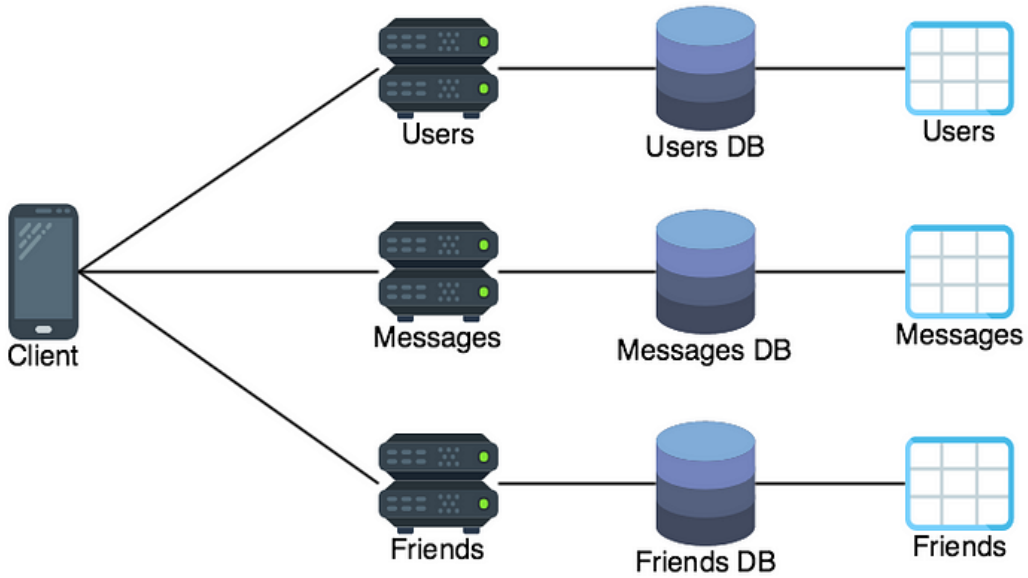
The shift from the traditional monolithic approach to separate databases for each microservice is an essential aspect of decentralisation of data management, a key principle in microservice architectures. In a traditional monolithic system, as show in Figure 3, multiple entities or data domains are being stored in a single, central database. This creates a tight dependency on the database. Any necessary change to the database schema or technology would potentially impact multiple parts of the application and unintentionally create side effects.

Figure 3. A monolithic application architecture using a central database (Peck, 2018b).



A microservices architecture, on the other hand and as already mentioned, promotes isolation and independence. For instance, as highlighted in the article by Nathan Peck, the team responsible for the development of the user service might opt for a NoSQL document store like MongoDB to handle flexible JSON schemas. Another team like the microservices team might opt for a relational database with SQL transactions to ensure data integrity, as seen in the imaginary game studio example.

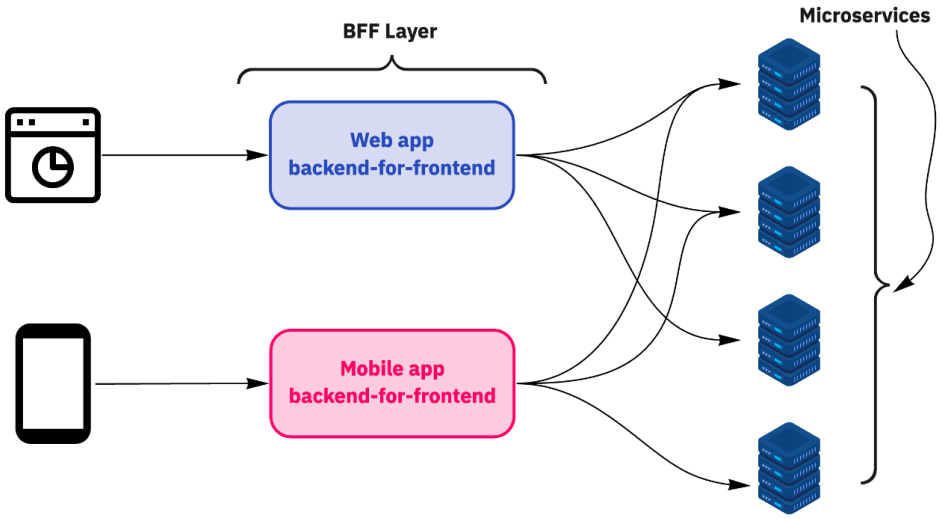
Figure 4. A microservices application using specialised databases (Peck, 2018b)



This decentralised approach enables polyglot persistence, a conceptual term that refers to using multiple, different data storage solutions within a single system (Wikipedia contributors, 2025). Teams are allowed to optimise each service for its specific set of tasks. Figure 4 visualises the concept of polyglot persistence. However, this approach introduces

its own set of complexities to the development process. When a transaction or query requires data from multiple sources (services), it becomes necessary to implement a mechanism for communication and data aggregation between services (Bhardwaj, 2024).

Figure 5. Diagram showcasing BFF pattern implementation (Bhardwaj, 2024).



This might involve API calls between services or potentially using patterns like the 'Backend for Frontends' (BFF) pattern, which allows composing data based on specific user requirements and needs (RobBagby, n.d.). Figure 5 demonstrates an implementation of the BFF pattern in microservices architecture.

3.3.6 High observability

With numerous independent services interacting with each other in a system, it is crucial to have real-time insights into the health and performance of the application. Monitoring allows us to track important metrics like CPU and RAM usage, response times, errors and request throughput for each service. This provides a view of the system's general status. (Virtuoso QA, n.d.)

Proactive monitoring can help identify potential problems before they evolve into bigger, mission-critical failures. By setting up alerts, teams can be notified of degraded performance, certain types of errors, or anomalies, which allows them to timely investigate and work on recovery (Swimm Team, 2024).

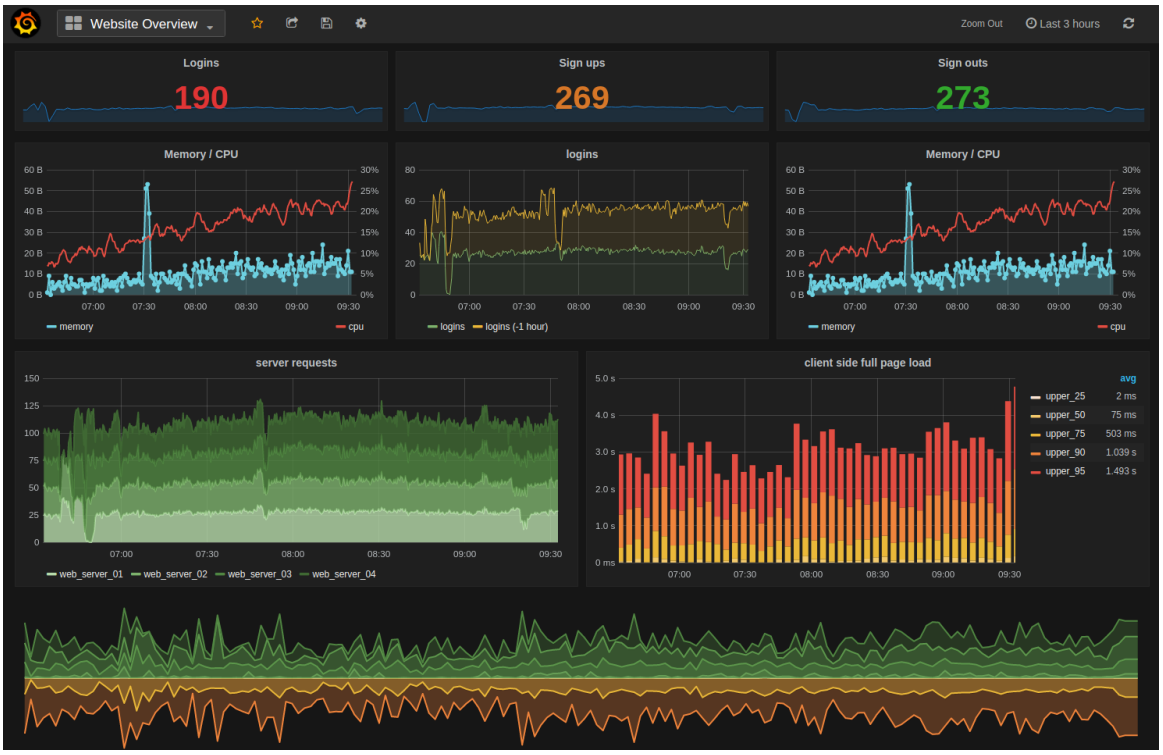
Monitoring also provides the data necessary to understand how the system performs under different workloads. Teams can analyse performance metrics to identify bottlenecks in the

system, optimise resource allocation, and make informed decisions about efficiently scaling the services that need more resources. This promotes optimal and effective use of infrastructure and helps planning capacity for future growth (Soni, 2025).

Monitoring logs and service activities, either centralised or on a stand-alone basis, can help detect suspicious behaviour or security breaches. Tracking API usage patterns can help identify unusual activity such as a sudden spike in requests from an unknown IP address, an abnormal number of failed log-in attempts on the user service, or access to sensitive endpoints that are normally not used by the client's front end (Atlassian, n.d.-c).

To effectively monitor microservices, popular open-source tools such as Prometheus, Grafana and Loki are frequently used together. Prometheus and its Operator collect and store metrics as time-series data, which is crucial for tracking performance, resource and health information.

Figure 6. Dashboard created with Grafana (Linux Screenshots, 2016)



Grafana provides powerful data visualisation capabilities, allowing users or system administrators to create comprehensive dashboards full of insightful graphs, by combining the metrics gathered by Prometheus and logs from Loki into a unified view. Figure 6 illustrates a dashboard created with Grafana.

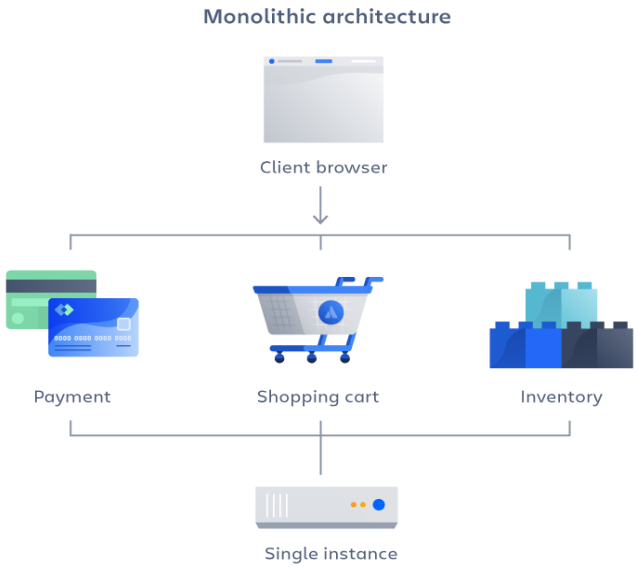
3.4 Comparison with other architectures

To gain a full understanding of the intricacies of microservices architecture, it is essential to consider its place within the wider context of software design. This chapter will provide a direct comparison between microservices and other significant architectural styles, with a particular focus on monolithic architecture and Service-Oriented Architecture (SOA). This analysis will highlight the key distinctions between them and the specific scenarios in which each architecture proves most effective.

3.4.1 Monolithic architecture

A monolithic architecture structures an application as a single, self-contained and inseparable entity, where the user interface (UI), business rules and data manipulation mechanisms are all part of the same codebase and deployment unit, as shown in Figure 7. In contrast, a microservices architecture divides the application into smaller, separately deployable services. Before the rise of microservices, software development was dominated by monolithic architectures, favoured for their simplicity and ease of deployment (GeeksforGeeks & navlaniwesr, 2024).

Figure 7. Diagram showcasing monolithic architecture (Atlassian, n.d.-c).



A monolithic architecture is characterised by its tightly integrated nature, where all parts of an application's functionality is developed, deployed and scaled as a single unit (AWS, n.d.-

b). This usually involves a unified code base that includes the UI, business logic and the data access layer (DAL) (Tanner, 2024).

All components within a monolithic application often share the same resources such as CPU, memory, database connections, which may lead to intrinsic dependencies between the different parts of the application, also known as tight coupling (GeeksforGeeks & navlaniwesr, 2024).

Although this approach may facilitate the initial development and deployment processes for smaller applications, it often brings challenges in terms of scalability, maintainability and technological flexibility as the application becomes more complex (Atlassian, n.d.-c).

In order to scale a monolithic application, it is usually necessary to scale the application as a whole, even if only a specific component experiences higher load (Tanner, 2024).

In addition, the entire application is usually built using a single technology stack (e.g. PHP for the back end of a web application), which limits the ability to adopt newer or more appropriate technologies for specific functionality as user and business requirements evolve (AWS, n.d.-b). As demonstrated in Table 1, Microservices and monolithic architectures are two different approaches to building applications. A monolith combines all features into one entity, whereas microservices divide an application into multiple smaller, standalone services (Solo.io, n.d.).

Table 1. A comparison between monolithic and microservices architectures

Monolith	**Microservices**
A single, tightly coupled application unit	Composed of multiple, independently deployable services
All components (UI, business logic, data access) are bundled together	Each service focuses on a specific business capability
Typically built using a single technology stack	Allows for the use of different technologies for different services

Failure in one component can potentially affect the entire system	Failure of one service is typically isolated and does not bring down the entire application
Can be simpler to develop and deploy initially for smaller applications	Requires planning beforehand and careful system design

3.4.2 Service-Oriented architecture (SOA)

Service-Oriented Architecture (SOA) is an architectural style which organises an application as a set of loosely connected services that communicate with each other, often using a shared communication infrastructure such as an Enterprise Service Bus (ESB) (Jamesmontemagno, 2022b; Red Hat, 2020). These services typically expose well-defined http service interfaces, often based on standards like SOAP or Representational state transfer (REST), allowing for interoperability between different systems and technologies (Oracle, n.d.).

A key characteristic of SOA is the emphasis on service reusability. In other words, services are designed to be discoverable and accessible for various applications within an enterprise (Oracle, n.d.). Service discovery is a vital component in SOA, facilitating the identification and interaction between services. Although SOA services aim for loose coupling, the use of a central ESB has the potential to introduce dependencies and complexity (Kong, n.d.-b).

SOA and microservices both aim for a service-based approach. However, there are significant differences between them. One important distinction lies in the granularity – read scope – of services (Powell, 2021). Microservices tend to be much smaller and more focused on specific business requirements (AWS, n.d.-c). In contrast, SOA services can be larger and include more functionalities. Another major difference is the approach concerned with resource sharing. Microservices typically have their separate and dedicated resources, including databases (Richardson, n.d.). Although SOA aims for loose coupling, it might involve services sharing certain infrastructure components managed by the ESB (Richardson & F5, 2015). Possible technology stack choices also differ; microservices enable polyglot persistence and programming, what allows teams to choose the best technology stack for each service, SOA, however, might lean towards more standardised enterprise-level technologies, usually determined by the ESB. SOA communicates differently as well. It relies heavily on a centralised and more heavyweight ESB, while

microservices often favour lightweight protocols like REST and event-driven communication (Richardson & F5, 2015).

3.5 Benefits of adopting microservices

The microservices architectural approach has the potential to transform how organisations build, deploy and manage applications. By dividing complex systems into smaller, independent services, companies can achieve a range of strategic benefits that address common challenges in modern software development. This approach enhances scalability and resilience, while enabling teams to innovate faster and more efficiently. In this chapter, we will explore ome of the key advantages of microservices architecture.

3.5.1 Improved scalability and elasticity

Microservices architecture provides substantial benefits in terms of scalability and elasticity when compared to monolithic systems (Atlassian, n.d.-c). In contrast to a monolithic system, where the entire application must be scaled even if only a specific component is under heavy load, microservices allow for the independent scaling of individual services based on their unique resource requirements (AWS, n.d.-d).

This level of scaling allows for more efficient resource utilisation. Organisations can efficiently allocate resources where they are needed, avoiding over-provisioning. For example, a media streaming application might have separate microservices for user authentication, video encoding and content delivery to the users. If there's a sudden increase in demand because users are trying to access more content, the content delivery service can be independently scaled to handle the increased workload, without needing other services to be scaled up as well (Eyevinn Technology, 2020; Atlassian, n.d.-c).

Furthermore, this independence in scalability enables better elasticity. The system can dynamically adapt to fluctuations in workload and demand. By scaling services up during peak times and down during periods of low demand, we can ensure optimal performance and cost effectiveness for organisations, which also contribute to sustainability by minimising energy consumption and energy waste. By only running the necessary resources to offer products organisations can lower their environmental impact and carbon footprint. Platforms like Kubernetes play a crucial role in managing this, providing orchestration capabilities to automate the scaling process based on demand, further optimising resource utilisation and contribute to more sustainable software operations (Tieturi, 2025).

3.5.2 Enhanced maintainability and testability

Adopting the microservices architectural model can help make applications much easier to maintain (Atlassian, n.d.-a). By dividing a large, complex system into smaller, independent services, each service becomes easier to understand, develop and maintain. This approach enables developers to specialise in specific functionalities without needing to comprehend the entire codebase, which is often the case with monolithic applications. This modularity simplifies debugging and allows for quicker identification and resolution of issues within a particular service (Kong, n.d.-a).

Additionally, changes and updates made to individual microservices can be implemented and deployed in a self-contained manner without affecting the functionality of other services. This reduces the risk associated with making changes to a large, tightly coupled codebase, where a modification in one area can have unintended consequences or side effects in others (AWS, n.d.-d). This enables teams to accelerate their iterative process and release new features or fixes more frequently for individual services, thereby fostering a more agile and responsive development environment (Atlassian, n.d.-a).

In terms of testability, there is also a definite improvement with microservices. Each service can be tested independently, making it easier to write unit and integration tests that focus on specific functionalities (Bhattacharjee, 2024). This isolation simplifies the testing process and allows for more thorough and efficient testing. Teams can apply different testing frameworks and strategies for each service, tailored to their specific needs (Mannotra, 2025).

This approach contrasts with the challenges of testing monolithic applications, where the interconnected nature of components can make it difficult to isolate and test specific parts of the system effectively. The ability to independently test and deploy microservices contributes to a more stable and reliable application overall.

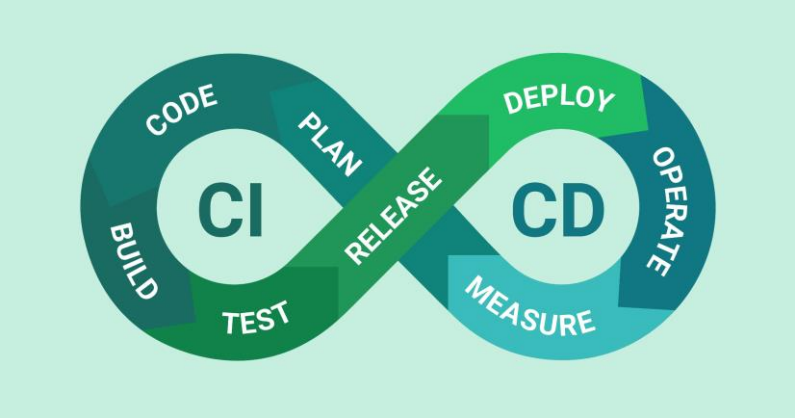
3.5.3 Increased development agility

Microservices promote increased development agility by organising development teams around specific business functions or individual services. Smaller, autonomous teams can work independently on their own microservices, allowing for parallel development efforts and reducing dependencies between teams. This autonomy empowers teams to make their own technology choices and manage their own development and deployment pipelines, leading to greater efficiency and faster decision-making.

Each microservice is self-contained, meaning it can be developed, tested and deployed independently of other services. This independent deployment helps achieve faster release cycles. Teams can release updates, new features and bug fixes for their specific microservice without needing to coordinate with other teams or redeploy the application in its entirety.

The ability to deploy applications more frequently and independently allows organisations to respond more quickly to user feedback and market competition. New features can be added incrementally, and bugs or issues can be fixed more quickly. This agility enables organisations to iterate faster on their applications, products and services, which helps them gain competitive edge and continuously deliver value to their users and customers.

Figure 8. Illustration of a CI/CD pipeline (Pandey, 2024).



Microservices enables such a streamlined application development and deployment process, often facilitated by implementing continuous integration and continuous delivery (CI/CD) pipelines, as show in Figure 8.

3.5.4 Better alignment with business capabilities

A microservices architecture has been shown to offer several key benefits, including a stronger alignment between the software development process and the organisation's core business capabilities (Palo Alto Networks, n.d.-a). In contrast to monolithic applications, for example, where functionalities are often tightly coupled and managed as a single entity, microservices enable the structuring of software around specific business domains or functions (Palo Alto Networks, n.d.-a). This alignment offers a multitude of benefits that contribute to enhanced agility, efficiency, and overall business value (Krishnan, 2024).

Microservices is an architectural style that deconstructs larger applications into a smaller and more independent set of services, with each one responsible for specific business capabilities. For instance, in an e-commerce platform, individual microservices might handle functionalities like product catalogue management, order processing, customer management, and payment processing. The direct mapping or linkage of software components to distinct business functions facilitates easier understanding, management and further refinement of the system by small, specialist development teams, which work in line with business needs (Bhattacharjee, 2025).

The team structure provides a deeper understanding of the business context of their services, enabling them to make more informed decisions and respond more efficiently to changing business needs (Elxecoraline, 2024). If a new business requirement arises, or an existing one changes, only the relevant microservice needs to be updated or modified. This targeted approach significantly reduces the complexity and risk associated with making changes to a large, monolithic codebase. As a result, organisations can adapt more quickly to market demands, customer feedback and competitive pressures, resulting in faster time to market for new features and enhancements (Bhattacharjee, 2025; GitLab, 2022).

Additionally, aligning microservices with business capabilities fosters clear ownership and accountability. Each team manages the entire lifecycle of their respective microservices – from development and deployment to maintenance and monitoring. This comprehensive responsibility encourages teams to take greater ownership of the reliability and performance of their services, directly benefiting the business functions they support and work in line with (GitLab, 2022).

3.6 Challenges and considerations of microservices

While the benefits of microservices are compelling, adopting this architectural style is not without complexities. This chapter will take a closer look at the various challenges and key considerations that organisations need to address when transitioning to or implementing a microservices architecture. It is important to understand these issues in order to successfully realise the benefits of microservices while mitigating the potential pitfalls.

3.6.1 Increased operational complexity

A key challenge introduced by the microservices architecture is the inherent increase in operational complexity compared to managing a monolithic application. Rather than dealing with a single, large deployable unit, operations teams now need to manage a multitude of

smaller and independent services. This increase in services requires a more sophisticated and dynamic infrastructure to handle provisioning, scaling and monitoring (Bhattacharjee, 2025). Each service, although independently manageable, adds to the overall operational burden and requires careful orchestration and management to ensure overall system coherence.

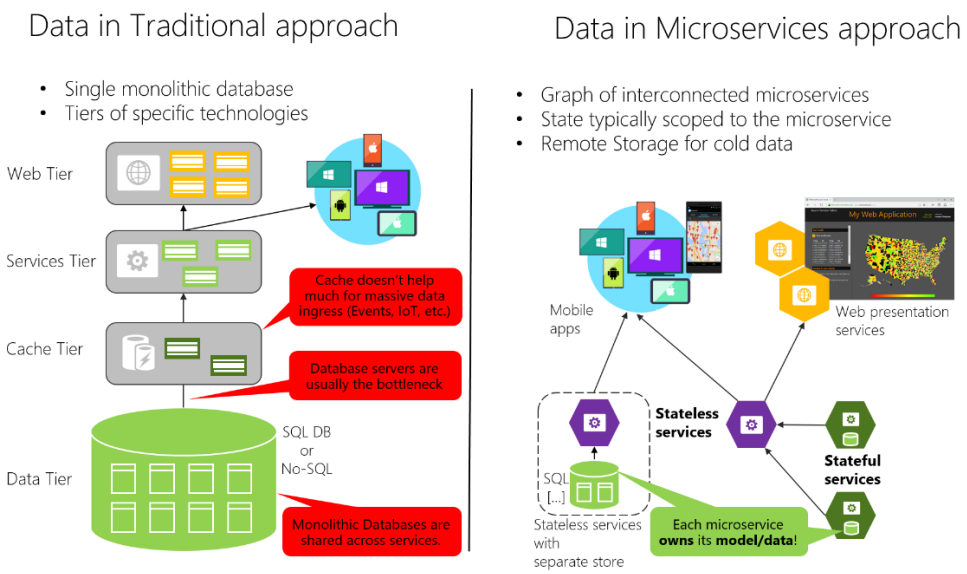
The deployment and orchestration of many independent services requires a high degree of automation. Manual deployment processes are impractical and prone to errors. The implementation and maintenance of CI/CD pipelines is therefore crucial for automating the build, test, and deployment processes for each microservice (Palachi, 2025). Container orchestration platforms like Kubernetes are often adopted to manage the lifecycle of these containers, handle scaling, and ensure high availability, adding another layer of technology and expertise required for operations teams (Palachi, 2025).

Configuring a distributed system with many services can also be significantly more complex than managing a single monolithic application. Each service may have its own set of configuration parameters, and ensuring consistency and proper rollout of configuration changes across all services requires careful planning and tooling (Elxecoraline, 2024). Service discovery, the mechanism by which services locate and communicate with each other, also adds to the operational burden, requiring reliable and scalable solutions to ensure smooth inter-service operations (Kong, n.d.-c).

3.6.2 Distributed system challenges

The microservices architecture is characterised by the division of an application into multiple independent services, which introduces a number of challenges typical of distributed systems (Design Gurus, n.d.). In contrast to monolithic applications, where components reside within the same process, microservices communicate over a network, introducing complexities related to reliability and latency. Network communication can be subject to delays, interruptions, and failures, requiring careful consideration in the design of inter-service interactions (Jamesmontemagno, 2022a). Figure 9 illustrates a data sovereignty in monolithic vs microservices architecture.

Figure 9. Data sovereignty comparison (Jamesmontemagno, 2022a).



Another significant challenge in distributed systems is maintaining data consistency across multiple independent services. Each microservice generally manages its own database, resulting in data silos. Ensuring data consistency across these databases can be complex, especially when transactions span multiple services (Awad, 2025). In such cases, traditional ACID (Atomicity, Consistency, Isolation, Durability) transactions become challenging to implement, often requiring the adoption of alternative approaches such as eventual consistency, where data may be temporarily inconsistent but will eventually reach a consistent state (Palle, 2024). This approach requires careful planning and consideration of its potential impact on the application's business logic.

In a distributed setting, coordinating operations across several independent services to ensure that either all operations succeed or none of them do (a core principle of atomicity) is significantly more complex. Solutions such as the Saga pattern, which involves a sequence of local transactions in each service with compensating transactions to undo changes if a failure occurs, are often employed to address this challenge (Awad, 2025).

In a microservices architecture, monitoring, logging and debugging are more complex. With numerous services potentially running on different hosts, correlating logs and tracing requests across service boundaries can be challenging (Fahmy, 2024). Therefore, comprehensive monitoring and logging strategies, in addition to distributed tracing tools, are essential to provide visibility into the health and performance of the overall system and to enable efficient diagnosis and resolution of problems as they happen.

3.6.3 Security considerations in a distributed environment

The transition from a monolithic architecture to a distributed microservices environment introduces a new set of security challenges that must be carefully addressed. In contrast to a single, self-contained application, a microservices-based system presents a considerably larger attack surface due to the large number of independent services communicating over a network (Okta, n.d.). Each service becomes a potential entry point for malicious actors, necessitating a comprehensive and layered security approach (Cloud Security Alliance [CSA], n.d.).

Ensuring the security of communication between individual microservices is essential. As services interact over a network, often using APIs, it becomes very important to implement robust authentication and authorisation mechanisms to ensure that only legitimate services can communicate with each other (Okta, n.d.). In addition, encryption of data in transit, using protocols like TLS/SSL, is essential to protect sensitive information from eavesdropping or tampering. Careful design around the security of the APIs themselves must also be considered, including input validation, rate limiting and protection against common web application vulnerabilities (Kong, n.d.-c).

Real-time security monitoring provides vital visibility into the health and behaviour of microservices. This allows for immediate detection of any anomalies or suspicious patterns that may indicate a security breach (Datadog, 2022). Comprehensive logging creates an auditable trail of events, which is crucial for investigating security incidents, understanding their impact, and implementing effective remediation strategies (Datadog, 2022). In the absence of robust monitoring and logging, identifying and responding to security threats in a complex, distributed microservices environment becomes significantly more challenging (Joyce & Kothamasu, 2024).

4 Continuous Integration and Continous Delivery (CI/CD)

Moving on from the architectural foundations of microservices, this chapter will focus on the crucial practices of continuous integration (CI) and continuous delivery (CD). These modern software development approaches aim to automate the processes of building, testing, and deploying software changes. CI/CD emphasises frequent code integration and the use of automated delivery pipelines. This enables development teams to receive rapid feedback, reduce errors, and release new features and updates faster and more reliably. Given the distributed and independent nature of microservices, a well-defined CI/CD pipeline is essential for their effective management and deployment, which we will explore in throughout this chapter.

4.1 History of the software development lifecycle (SDLC)

The software development journey has evolved significantly over the decades, driven by the increasing complexity of software systems and the growing demand for faster delivery cycles. In the early days of software engineering, the Waterfall model was a prevalent approach. This model characterised a linear, sequential flow with distinct phases such as requirements gathering, design, implementation, testing, and deployment (Seshadri, 2025). The Waterfall model's linear and sequential nature often led to extended development cycles and limited opportunities for feedback until late in the process, as each phase was typically completed before moving on to the next (Singh, 2025).

The Waterfall model offered a well-structured framework; however, its rigidity often made it difficult to adapt to the shifting requirements of projects. Changes identified at a later stage in the development cycle could be costly and time-consuming to implement. This lack of flexibility and the lack of early and frequent feedback led to the emergence of more iterative and agile methodologies (Singh, 2025).

Approaches such as the Iterative model and later Agile methodologies, including Scrum and Kanban, emphasised breaking down the development process into smaller, more manageable iterations (MCSGA, 2024). These methodologies focus on delivering working software at regular intervals, fostering collaboration between development teams and stakeholders, and adapting to changing requirements throughout the project lifecycle (MCSGA, 2024).

The core principles revolved around early and continuous feedback, allowing for the necessary course corrections to make sure that the delivered software better met user

needs (Laoyan, 2025). Figure 10 visually represents the key stages and principles of the Agile methodology.

Figure 10. Key stages in the Agile methodology (Nvisia Learn, 2023).

Software release frequencies have increased, and deployments have grown in complexity. The need for automation has arisen due to manual processes often resulting in errors and time delays, hindering consistent, rapid delivery of value. This has paved the way for practices focusing on automation, leading to Continuous Integration (CI) and Continuous Delivery (CD) as key components of the modern Software Development Lifecycle. CI/CD enables teams to frequently integrate code, automated testing and deliver software updates with greater speed and reliability, improving on manual methods (Red Hat, 2023b).

It is important to note that the Waterfall model was not the only paradigm explored in the early stages of software development. The V-model emerged as an extension of the Waterfall, emphasising the relationship between each development phase and a corresponding phase of testing. For instance, requirements gathering was linked to acceptance testing, and design to system testing (Wikipedia contributors, 2024). By contrast the Big Bang model was much less structured. Development typically began with little planning and resources, focusing on coding and seeing what emerged. While offering flexibility, this model was best suited to very small projects with very limited scope (Nazarenko, 2025).

4.2 Core concepts of CI/CD

Continuous Integration (CI) and Continuous Delivery (CD) are two fundamental practices in modern software development. The aim of these practices is to automate and streamline

the process of building, testing and releasing software. At its core, Continuous Integration focuses on frequently merging code changes from individual developers into a shared repository, ideally multiple times a day (Atlassian & Pittet, n.d.). This is coupled with an automated build process that compiles the codebase and runs a suite of automated tests, including unit tests, integration tests, and potentially more comprehensive end-to-end tests (Red Hat, 2023b).

The primary objective of CI is to identify and rectify integration issues and bugs at the earliest stage of the development cycle, thereby ensuring rapid feedback to developers and maintaining the integrity of the codebase (Atlassian & Pittet, n.d.). By automating these steps and encouraging frequent integration, CI minimises the chances of encountering significant integration problems later in the development process, which can be costly and time-consuming to fix (Harness & Gaikwad, 2024). Building on the foundation of CI, Continuous Delivery extends the automation to the software release process. While CI ensures that code changes are frequently integrated and tested, CD focuses on automating the steps required to release the software to various environments, including staging and production (Atlassian & Pittet, n.d.). This involves the automation of deployment processes, configuration management and potentially even infrastructure provisioning. Continuous Delivery's core principle is to ensure that software is always in a deployable state, enabling rapid and reliable releases whenever business demands dictate (Atlassian & Pittet, n.d.).

In many organisations, this process is extended to Continuous Deployment, where every code change that passes through all stages of the pipeline is automatically deployed to production. Irrespective of the term used - Continuous Delivery or Continuous Deployment - the primary objective is to accelerate the delivery of new features and updates to users, mitigate the risk associated with releases and enable faster time-to-market for software products (Palo Alto Networks, n.d.-b). CI/CD is essentially a culture and set of practices that promote automation, collaboration and frequent feedback throughout the software development and delivery lifecycle (Red Hat, 2023b).

4.3 Common tools in CI/CD Pipelines

Implementing a production-ready CI/CD pipeline often involves the use of various specialised tools that automate different stages of the software development and delivery process. These tools can be categorised according to their primary function within the pipeline. Version control systems, such as Git, often hosted on platforms like GitHub, GitLab, or Bitbucket, form the foundation of CI by managing code changes and facilitating

collaboration among developers (Das, 2024). These platforms also frequently offer built-in CI/CD capabilities, such as GitHub Actions (GitHub, n.d.-a).

In the context of build automation, a range of tools is available, including Jenkins, GitLab CI/CD, GitHub Actions, CircleCI and Travis CI. These tools automate the process of compiling code, running static analysis, and packaging the application into deployable artefacts. These tools are typically triggered by code commits to the version control system and provide feedback on the success or failure of the build process (Ndungu, 2022).

Testing is an invaluable part of CI/CD, and different tools are used to automate different types of testing. Popular frameworks for unit testing include JUnit and Mocha, while Selenium and Cypress are tools of choice for automated browser testing (Dwyer, 2017). In addition, API testing tools and performance testing tools play a vital role in ensuring the quality and reliability of the software (Dwyer, 2017).

Once the build and testing phases have been successfully completed, artefact management tools are used to store and manage the resulting build artefacts, such as compiled binaries, container images, or configuration files (Harness & Minick, 2024). These tools provide a centralised repository for versioned artefacts, making them easily accessible for deployment (Tetteh, 2025).

The deployment phase of the CD pipeline often utilizes tools like Ansible, Terraform and Puppet for infrastructure provisioning and configuration management. In containerised applications, Kubernetes and Docker are frequently used for orchestrating and deploying containers across different environments. These tools automate the deployment process to staging, testing and production environments, which ensures consistency and reducing manual errors (Tetteh, 2025).

5 Kubernetes

In the current software development environment, the ability to deploy, manage, and scale applications efficiently and reliably is important (Nayak, 2025). Kubernetes, often shortened as “k8s”, has become the de facto standard for container orchestration, providing a robust and extensible platform that addresses the complexities of modern, distributed systems, particularly those built using a microservices architecture (Google Cloud, n.d.). This chapter will cover the essential concepts of Kubernetes, explain how it operates, highlight its self-healing and resilience features, review the various distributions available, and show how it can be integrated smoothly with Continuous Integration and Delivery (CI/CD) pipelines, emphasising its crucial role in enhancing application reliability and scalability.

5.1 Core concepts of Kubernetes

At its core, Kubernetes introduces several key abstractions that define the desired state and structure of an application deployment. Pods are the most basic unit of deployment, encapsulating one or more containers that are tightly coupled and share a common set of resources, such as network interfaces and storage volumes (Ricky, 2023). This co-location and resource sharing within a Pod enables closely related containers to function as a cohesive unit, often representing a single logical component of an application.

In Kubernetes, pods are executed on nodes, which function as the worker machines in a cluster. These nodes can be physical servers or virtual machines, each one hosting multiple Pods and providing essential resources, including compute, memory, and network capacity (Ricky, 2023). A collection of these nodes, along with the control plane, forms a cluster, which is the complete operational environment managed by Kubernetes.

To provide logical isolation and resource management within a cluster, Kubernetes uses namespaces (Manasa, 2024). These virtual clusters allow multiple teams or projects to operate independently within the same physical cluster, preventing resource name collisions and enabling fine-grained access control.

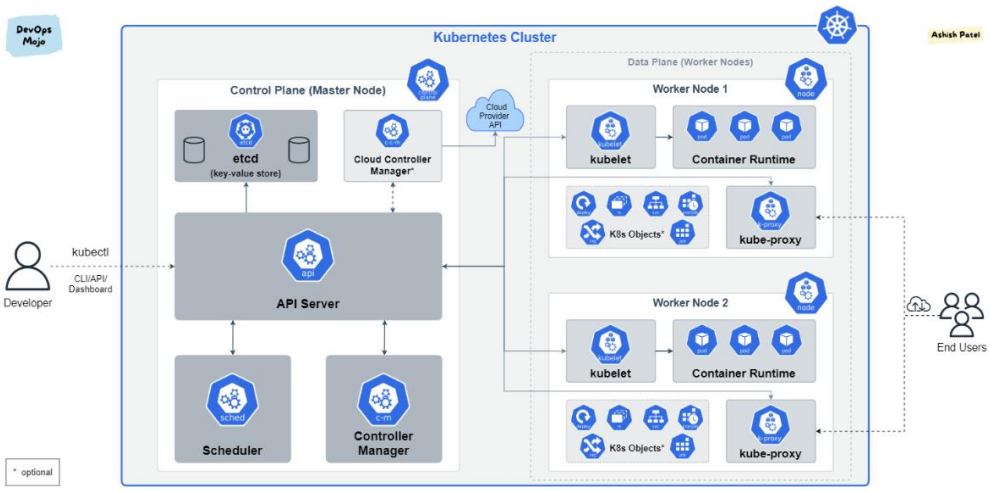
Kubernetes utilises higher-level abstractions, such as Deployments, to manage the lifecycle of Pods and to ensure that the desired number of instances are running. A Deployment defines the desired state for a set of identical Pods, managing the underlying ReplicaSets. These ensure that the specified number of Pod replicas are always available (Spot.Io, 2024). This abstraction simplifies the process of updating applications and scaling them up or down.

Services offer a reliable abstraction that enables access to a set of Pods. Rather than directly interacting with the ephemeral IP addresses of individual Pods, which can change as they are created or destroyed, clients interact with a Service (Spot.Io, 2024). Kubernetes's automatic traffic routing to healthy Pods backing the Service provides load balancing and ensures application availability even as the underlying Pods change.

5.2 How Kubernetes works

The distributed architecture of Kubernetes is comprised of a central control plane which oversees the cluster as a whole and worker nodes on which application workloads are executed. The control plane acts as the brain of the Kubernetes cluster, making global decisions about the cluster and detecting and responding to events. Its key components include the API Server, which serves as the central point of interaction for all administrative tasks and component communication via the Kubernetes API (Spot.Io, 2024). Etcd is a distributed, reliable key-value store that functions as Kubernetes' single source of truth (Mr.PlanB, 2024). It stores the configuration data, the state of the cluster, and the metadata. The Scheduler's primary function is to intelligently allocate newly created Pods to the most suitable Nodes, considering resource availability, constraints, and established policies (Ricky, 2023). Finally, the Controller Manager runs a suite of controller processes that continuously monitor the state of the cluster and work to bring the current state into the desired state defined by the user. For instance, the Deployment Controller ensures the number of Pod replicas running aligns with the set target, while the Node Controller oversees the health status of the worker nodes. Worker nodes are where the application containers are run. Each node is host to a kubelet, an agent that communicates with the control plane, ensuring the Pods running on that specific node are managed and the containers defined within the Pod specification are running and healthy (Manasa, 2024). The kube-proxy is a network proxy that runs on each node and implements the Kubernetes Service abstraction. The kube-proxy then manages network rules to forward traffic from Services to the correct backend Pods, ensuring load balancing and service discovery within the cluster (Manasa, 2024). The operational flow generally starts with a user defining the desired state of the application using the Kubernetes API. This state is then stored in etcd. The scheduler then assigns the necessary Pods to available nodes.

Figure 11. Kubernetes' architecture and internal components (Patel, 2024).



The kubelets on these nodes then receive instructions to launch the containers. The controller manager is responsible for monitoring the cluster state continuously and taking corrective actions if the actual state deviates from the desired state (Prajapati, 2024). This ensures the overall health and stability of the applications. Figure 11 illustrates a diagram of Kubernetes' internal architecture and components.

5.3 Inherent self-healing mechanisms

A key benefit of Kubernetes is its inherent capacity to automatically identify and recover from failures, thereby enhancing the resilience of applications. This self-healing capability is primarily achieved through (The Kubernetes Authors, 2025):

Health Checks (Liveness and Readiness Probes): Kubernetes enables developers to define probes that periodically verify the health and readiness of containers within a Pod. Liveness probes determine if a container is running correctly and should be restarted if it is unhealthy. This is crucial for recovering from deadlocks or other internal container failures. Conversely, readiness probes ascertain if a container is prepared to receive traffic. If a Pod fails a readiness probe, Kubernetes will halt the routing of traffic to it until it recovers, thus preventing users from accessing unhealthy instances. Restart Policies: When defining a Pod, a restart policy (Always, OnFailure, Never) can be specified to dictate how Kubernetes should handle container exits. For most production workloads, the "Always" policy is used to ensure that containers are automatically restarted if they exit due to a failure. Controllers: The primary function of these controllers (Replication controllers, ReplicaSets, Deployments) is to continuously monitor the number of active Pod replicas for a specified application. In the event of a Pod failure or a node becoming unavailable, the

controller detects this discrepancy and automatically creates new Pod instances on healthy nodes to maintain the desired number of replicas. As a result, the application remains available even if the underlying infrastructure fails.

5.4 Enhancing resilience beyond self-healing

In addition to its core self-healing functionality, Kubernetes provides a range of features that contribute to the development of highly resilient applications (Anand, 2025): Rolling Updates and Rollbacks: Deployments facilitate seamless application updates with minimal downtime through rolling updates, where new versions of Pods are gradually rolled out while old versions are phased out. In the event that a new version introduces issues, Kubernetes allows for straightforward rollbacks to the previous stable version, thereby minimising the impact of any faulty deployments. Resource Management (Requests and Limits): By defining resource requirements (the minimum resources guaranteed to a Pod) and limits (the maximum resources a Pod can consume), developers can make sure that applications have the resources they need to function properly and prevent resource bottlenecks that could lead to instability. Affinity and Anti-Affinity Rules: Affinity rules allow you to specify that certain pods should be scheduled on the same node or set of nodes. Anti-affinity rules enable you to ensure that pods belonging to the same application are spread across different nodes or availability zones. This helps to improve fault tolerance by preventing a single point of failure from taking down all instances of an application.

5.5 The diversity of Kubernetes distributions

Although the foundational technology is supplied by the core Kubernetes project, various distributions are available to cater to different deployment environments and user needs. Vanilla Kubernetes represents the upstream, open-source project, offering maximum flexibility but requiring more manual configuration and management. For local development and testing, lightweight distributions such as Minikube and kind provide a single-node Kubernetes cluster on a developer's machine (Kubecost, n.d.). Cloud providers offer fully managed Kubernetes services.

Figure 12. Logos of various Kubernetes distributions (Krzywiec, 2020).

These include Amazon Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE) and Azure Kubernetes Service (AKS). These services abstract away the complexities of control plane management, providing a more streamlined and scalable experience. Distributions such as Rancher and OpenShift offer supplementary features and tools, with a particular emphasis on enterprise-grade security, management, and developer experience (Kubecost, n.d.). The choice of distribution depends on factors such as the scale of the deployment, the level of management desired, and specific organisational requirements (Nuageup Team, 2024). As demonstrated in Figure 12, there is a wide variety of logos representing various Kubernetes distributions.

5.6 Scalability and Kubernetes

Kubernetes is designed to enable applications to scale effectively at both the application instance level and the underlying infrastructure level. This inherent scalability allows systems to adapt to varying demands, ensuring optimal performance and resource utilisation.

To manage application-level scaling, Kubernetes uses the Horizontal Pod Autoscaler (HPA). The HPA automatically adjusts the number of Pod replicas for a given Deployment or ReplicaSet based on observed metrics, most commonly CPU utilisation. Should the defined metric exceed a predetermined threshold, the HPA will automatically increase the number of running Pods to handle the increased load. In contrast, if the metric falls below a certain threshold, the HPA can scale down the number of Pods, thereby optimising resource consumption during periods of lower demand (Dass, 2024). This dynamic scaling

ensures that applications can efficiently handle fluctuating traffic without manual intervention.

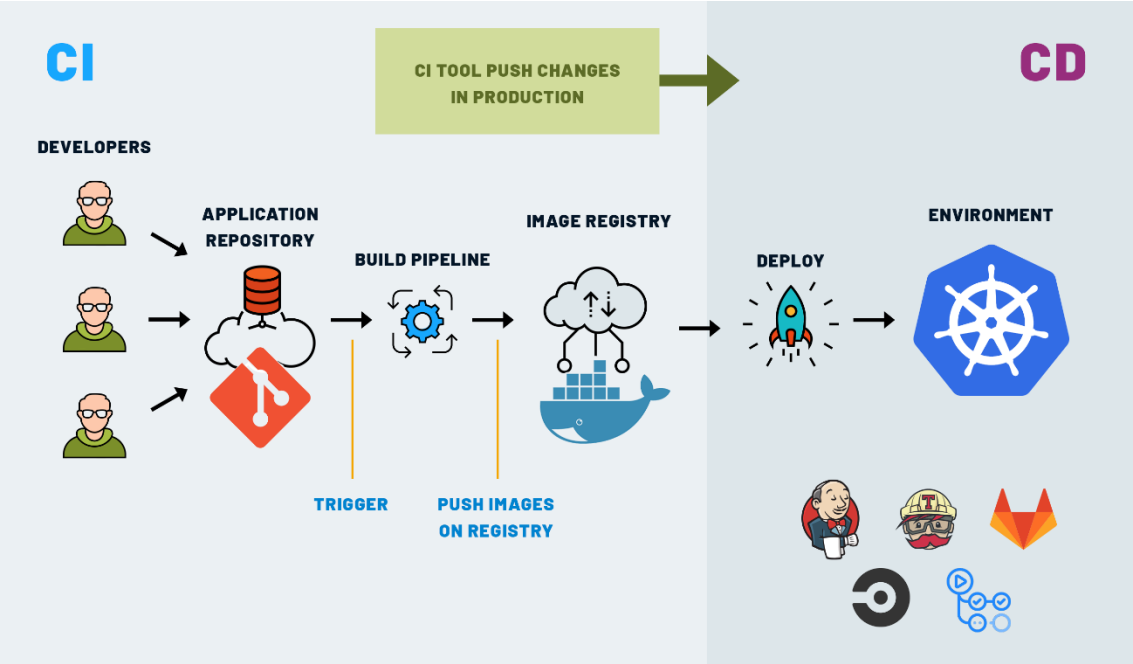
In addition to horizontal scaling, Kubernetes also offers capabilities for Vertical Pod Autoscaling (VPA). The VPA, in contrast to the HPA, which adjusts the number of pods, focuses on adjusting the resources allocated to individual pods, specifically CPU and memory (Muppeda, 2024). By analysing the historical and current usage of resources, the VPA can make recommendations or apply changes to resource requests and limits automatically (Muppeda, 2024). This ensures that Pods are appropriately provisioned, avoiding unnecessary over-provisioning and resulting in better overall resource utilisation within the cluster.

Kubernetes has the capability to scale the underlying infrastructure via the Cluster Autoscaler. The Cluster Autoscaler integrates with cloud provider APIs to automatically adjust the number of worker nodes in the Kubernetes cluster based on the resource requirements of the running Pods (Tripathy, 2021). In instances where certain Pods cannot be scheduled due to insufficient resources, the Cluster Autoscaler will provision new nodes. Conversely, if nodes are not fully utilised for an extended period of time, the cluster can be scaled down by removing nodes (Tripathy, 2021). This adaptive scaling ensures that the cluster has the necessary capacity to run the applications while also optimizing costs by avoiding unnecessary resource allocation.

5.7 Integration with CI/CD pipelines

Kubernetes integrates fully with CI/CD pipelines, offering a fully automated workflow from code commit to production deployment of resilient applications (Dwyer, 2023). Container registries play a central role, as CI/CD pipelines typically build Docker images of applications and push them to registries like Docker Hub or cloud-specific registries. Kubernetes then pulls these images to run the application containers. CI/CD pipelines can take advantage of Kubernetes Deployment Strategies, including rolling updates and canary deployments, to ensure the release of new application versions can be done with minimal disruption to users (Dwyer, 2023).

Figure 13. Illustration of a push-based CI/CD pipeline deploying to Kubernetes (Sparkfabrik, 2021).



Kubernetes configuration is declarative in nature, defined in YAML or JSON files (Peefy, 2024). This allows CI/CD systems to manage and version-control application deployments. Many well-known CI/CD tools offer native integration or plugins for interacting with Kubernetes clusters, simplifying tasks such as deploying new versions or performing rollbacks within the cluster (Dwyer, 2023). Figure 13 shows a typical push-based CI/CD pipeline where developers commit code, which then triggers a build process, container images built are then pushed to a registry and subsequently deployed to a Kubernetes cluster.

6 Chaos engineering

Following on from the discussion of continuous integration and delivery as a means of efficiently deploying microservices, the next important aspect of creating robust and scalable applications is ensuring their resilience. This is where the discipline of Chaos Engineering comes in. Rather than waiting for failures to occur in production, Chaos Engineering proactively introduces controlled experiments to identify weaknesses and failure points within a system before they cause widespread outages. This chapter will focus on the core concepts, benefits, and practical application of Chaos Engineering, particularly within the context of microservices architectures.

6.1 Core concepts of chaos engineering

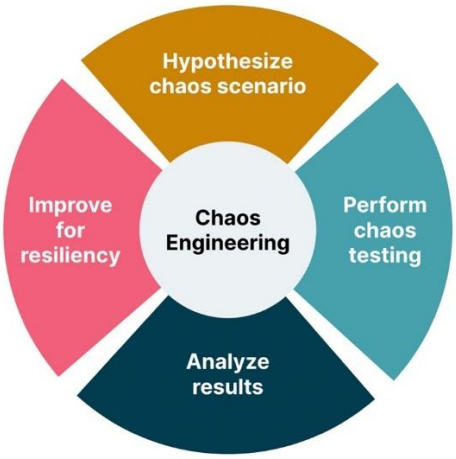
At the heart of Chaos Engineering lies the proactive introduction of controlled failures into a production system to build confidence in the system's ability to withstand turbulent conditions (Gunja, 2024). This discipline operates on the principle that by deliberately causing failures, weaknesses within the system can be uncovered before they manifest as widespread outages impacting users (Gremlin, 2023b). A fundamental concept in Chaos Engineering is the formulation of hypotheses around the "steady state" of a system. The steady state represents the normal, expected behaviour of the system under typical operating conditions, often defined by measurable metrics such as throughput, latency, error rates, and resource utilization (Gremlin, 2023b).

Chaos experiments are designed to simulate real-world failures that a system might encounter in production. These scenarios can range from server crashes and network latency to resource exhaustion and unexpected traffic surges (Gremlin, 2023b). The emphasis is on emulating realistic failure scenarios rather than introducing arbitrary faults. These experiments are not conducted randomly; they are controlled experiments with clearly defined objectives and a blast radius to minimize potential negative impact on users (Steadybit, 2024a). While these experiments are often performed in production environments to gain the most accurate insights, safeguards and monitoring are essential to contain any unintended consequences (Steadybit, 2024b).

To gain meaningful insights and ensure continuous improvement, it is recommended that chaos experiments be automated (Gremlin, 2023b). Automation facilitates the regular and reproducible execution of experiments, enabling teams to continuously validate the resilience of their systems as changes are introduced. The impact of these experiments is then meticulously measured against the initially defined steady-state hypothesis. Should there be any deviations from the expected behaviour, this would indicate potential weaknesses or vulnerabilities within the system that need to be addressed (Darktrace, n.d.).

The objective of chaos engineering is to learn and improve. By observing how the system reacts to injected failures, development and operations teams gain valuable insights into its reliability and fault tolerance characteristics (Sapkota, 2024). This knowledge informs efforts to strengthen the system, improve its fault tolerance, and enhance its overall reliability and stability. This iterative process of experimenting, observing, and remediating is central to building truly resilient distributed systems (Wickramasinghe, 2024).

Figure 14. Core Concepts in the Chaos Engineering Process (Parekh & Ramakrishnan, 2023).



As illustrated by Figure 14, Chaos Engineering can be viewed as a cyclical process. It begins with formulating a hypothesis about the expected behaviour or steady state of the system. Following this, testing involves designing and executing controlled experiments to validate this hypothesis by injecting failures (Wickramasinghe, 2024). A key element of this testing phase is defining and controlling the blast radius, which limits the scope and potential impact of the experiment. Finally, insights gained from this process inform improvements and potentially generate new hypotheses for further experimentation (Gunja, 2024). This iterative cycle of hypothesising, testing, limiting impact, and gaining insights is central to the practice of Chaos Engineering.

6.2 Benefits of chaos engineering

The adoption of Chaos Engineering can provide organisations with a number of significant benefits when it comes to building and maintaining reliable and robust software systems. Foremost among these is the enhancement of system resilience. By proactively introducing controlled failures into a production environment, teams can identify hidden weaknesses and vulnerabilities that may not be detected in standard testing scenarios (Parekh & Ramakrishnan, 2023). This proactive approach enables the implementation of fixes and improvements before these weaknesses can lead to costly and disruptive outages for end-users.

Additionally, it fosters increased confidence among development and operational teams in their system's ability to effectively manage unanticipated events. By subjecting the system to a range of failure scenarios and observing its recovery mechanisms in a controlled

setting, teams can develop a more profound understanding and confidence in the system's stability. This proactive approach ultimately leads to reduced downtime in production (AWS, n.d.-d). By addressing potential failure points early on, businesses can reduce the frequency and duration of service interruptions, ensuring business continuity and protecting revenue (Corp, 2024a; Corp, 2024b).

Chaos Engineering also provides a more in-depth understanding of complex system behaviour. Distributed systems, especially those built with microservices, can exhibit intricate and often unpredictable interactions. Chaos experiments help to illuminate these behaviours under stress, revealing dependencies and failure modes that might otherwise remain unknown (Gunja, 2024). This enhanced awareness enables teams to engineer more resilient architectures and implement more effective monitoring and alerting strategies.

The benefits of Chaos Engineering are reflected in improved customer satisfaction. Reliable and stable systems deliver better user experiences and increased confidence in the organisation's services (Moropo, 2023). Chaos Engineering can also contribute to significant cost savings by reducing downtime and improving operational efficiency, minimising the financial impact of production outages and optimising resource allocation. In essence, Chaos Engineering shifts the focus from reactive incident response to proactive resilience building, resulting in more dependable and robust software systems (Gremlin, 2023b).

6.3 Challenges of chaos engineering

Although the advantages of Chaos Engineering are considerable, implementing it can be challenging. Organisations embarking on this journey often encounter several hurdles that need careful consideration and strategic planning. A key challenge is the cultural shift required to embrace the intentional creation of failures in production environments, which often requires a mindset shift away from a fear-based approach (Arcot and Vitucci, n.d.). This necessitates a shift in mindset, emphasising the learning and improvement aspects of Chaos Engineering, as opposed to a "breaking things" mentality (AWS, n.d.-a).

Another significant challenge arises from the complexity of modern distributed systems, particularly those built using microservices. Designing and executing meaningful chaos experiments in such intricate environments can be difficult. Identifying the right types of failures to inject, understanding the potential cascading effects, and accurately attributing

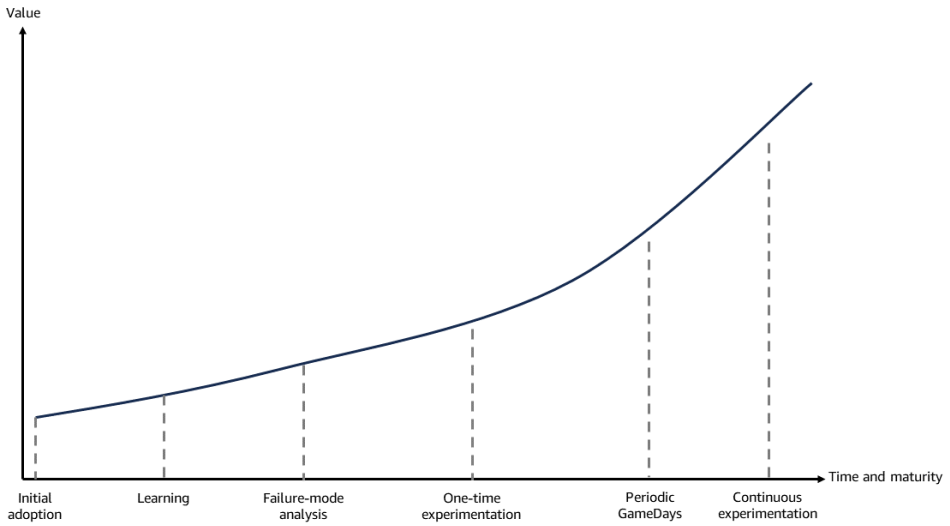
observed behaviour to the experiment require a deep understanding of the system's architecture and dependencies (Gunja, 2024; Arcot and Vitucci, n.d.).

Effective Chaos Engineering heavily relies on mature monitoring and observability capabilities (Gunja, 2024; Wickramasinghe 2023). Without comprehensive visibility into the system's behaviour, it becomes challenging to accurately define the "steady state" – the normal operating condition against which the impact of experiments is measured. Furthermore, interpreting the results of chaos experiments requires detailed metrics and logs to understand the root causes of any observed deviations (Wickramasinghe, 2023).

Conducting effective chaos experiments demands a specific skillset and expertise (AWS, n.d.-d). Teams need to develop the ability to formulate meaningful hypotheses, design impactful experiments, safely execute them, and analyse the resulting data to drive improvements (Arcot and Vitucci, n.d.). This may require investment in training or the recruitment of individuals with experience in distributed systems and resilience testing.

Organizational culture and potential resistance to intentionally breaking production systems can also pose a significant challenge (AWS, n.d.-a; Arcot and Vitucci, n.d.). Some teams may be hesitant or fearful of causing disruptions, even in a controlled manner. Overcoming this resistance requires a shift in mindset, emphasizing the proactive nature of Chaos Engineering as a way to prevent future outages and improve overall system reliability. Figure 15 illustrates how value increases as chaos engineering adoption progresses over time.

Figure 15. Increasing Value Over Time with Chaos Adoption (AWS, n.d.-a).



Finally, it is important to recognise the challenges organisations may face when accurately defining a steady state for complex systems and selecting the most impactful experiments to run (Gunja, 2024). Understanding the intricate dependencies and failure modes within a distributed environment requires careful analysis and a deep knowledge of the system's architecture.

6.4 Common tools and techniques

The implementation of Chaos Engineering often relies on specific tools to inject failures into a system. While several tools exist, a few have risen to prominence within the IT industry.

One of the earliest and most well-known is Netflix Chaos Monkey. Originally developed by Netflix, this tool randomly terminates virtual machine instances within their production environment to ensure the resilience of their services. Its simplicity and effectiveness in uncovering single points of failure made it a foundational tool in the Chaos Engineering movement (Schillerstrom, 2022).

Chaos Mesh, on the other hand, is a cloud-native Chaos Engineering platform specifically designed for Kubernetes environments. It allows for a wide range of fault injections at different levels, including pod failures, network disruptions, and even kernel-level faults (Chaos Mesh, n.d.). This makes it particularly relevant for modern microservices architectures.

Another popular open-source tool is Litmus Chaos. Litmus provides a framework for creating, managing, and monitoring chaos experiments in Kubernetes. It offers a library of pre-built chaos experiments and allows users to define custom scenarios using a declarative approach (Schillerstrom, 2022). Its emphasis on a collaborative approach to Chaos Engineering enables teams to easily share and reuse experiments (Litmus Chaos, n.d.).

Gremlin is a commercial Chaos Engineering platform that offers a range of fault injection capabilities across various infrastructure types, including cloud, on-premise, and hybrid environments. It provides a user-friendly interface and focuses on safety and control, allowing teams to gradually introduce failures and monitor their impact in a controlled manner (Schillerstrom, 2022).

It should be noted that these tools represent some of the most prominent options currently available; however, the landscape of Chaos Engineering tools is continuously evolving with

new solutions emerging. Beyond specific tools, common techniques in Chaos Engineering include latency injection to simulate network delays, resource exhaustion to test how systems behave under heavy load, and stateful service disruption to evaluate the resilience of databases and other stateful components (Gremlin, 2023b). These techniques, often facilitated by the tools mentioned, help organisations proactively identify and address weaknesses in their systems, ultimately leading to more reliable and resilient applications.

6.5 The business case for chaos engineering

In today's digital world, businesses rely heavily on the availability and reliability of their software systems. Outages and performance issues can result in serious financial loss, damage to brand reputation and erosion of customer confidence (Parashar, 2024). While traditional testing methods are valuable, they often lack the capacity to identify complex and unexpected failures that can occur in distributed systems operating at scale (Parashar, 2024). Chaos Engineering, utilising a scientific and proactive approach, provides a solution by strengthening systems and enhancing resilience, delivering substantial benefits to businesses (Hicks, 2025).

The business case for Chaos Engineering is based on the principle of risk mitigation and business continuity. Intentionally introducing failures into a production environment helps organisations proactively identify weaknesses and vulnerabilities before they impact users and revenue (IBM, 2023). This proactive approach is in stark contrast to reactive incident response, which is often more costly and disruptive.

A primary benefit of Chaos Engineering for businesses is a reduction in downtime and associated costs. The impact of unplanned outages on revenue is often direct and substantial, especially for businesses that rely on online transactions or service availability (AWS, n.d.-d; Nutanix, 2020). Organisations can reduce the likelihood and duration of such outages by identifying and addressing potential failure points through controlled experimentation, leading to substantial cost savings. To give an example, an e-commerce platform that experiences frequent downtime during peak periods will directly benefit from the improved stability and availability that chaos engineering promotes. Another example would be an ATM machine. Picture a scenario where a customer attempts to withdraw cash from an ATM and encounters a failure before receiving their money. This would cause the customer concern, and the bank would suffer a negative reputation.

Chaos Engineering helps increase customer satisfaction and loyalty (Nutanix, 2020). Customers demand a smooth and reliable experience. Frequent disruptions or substandard

performance can lead to customer frustration and eventually drive customers to competitors (AWS, n.d.-d). By implementing measures to ensure the resilience of their systems, businesses can deliver a more consistent and positive user experience, leading to increased customer retention and positive word-of-mouth. This is particularly important in highly competitive markets where even minor disruptions can have a significant impact on customer perception.

Beyond the prevention of unfavourable scenarios, chaos engineering also enables faster innovation and delivery of new features (Gunja, 2023). When development teams have greater confidence in the stability and resilience of their underlying systems, they can iterate and release new features more rapidly and with less fear of introducing critical failures. Being more agile can be a significant competitive advantage, allowing organisations and businesses to respond quickly to market demands and customer needs (Udasi, 2024). The insights we gain from chaos experimentation inform architectural decisions and development practices, leading to more robust and resilient designs from the very beginning (Gunja, 2023; Goikhman, 2024).

Applying chaos engineering also fosters a deeper understanding of system behaviour and dependencies across teams. Through collaborative design and experimentation, development, operations and security teams gain critical insight into how different components of the system interact under stress and potential failure modes (Wickramasinghe, 2023). This shared knowledge improves communication and collaboration and ultimately leads to more effective problem solving and incident response when real problems arise (Goikhman, 2024).

The AWS S3 outage in 2017 is a prime example of how a seemingly minor configuration error can cascade into a widespread system disruption. The incident was caused by a misconfigured update to the S3 service that changed critical network routing and configuration parameters, inadvertently causing internal systems to misinterpret service health signals (Newton, 2017; Helmke, 2019). The misconfiguration resulted in the removal of certain S3 endpoints from active service, causing traffic to be rerouted through fallback systems which were not designed to handle the sudden increase in traffic (Helmke, 2019). As a result, the cascading effect of increased latency and intermittent outages began to impact not only S3 itself, but also a host of dependent services throughout the AWS ecosystem (Novet, 2017; Helmke, 2019).

The outage had a significant ripple effect, causing widespread disruption to services that rely on S3 for data storage and retrieval. These services ranged from internal AWS tools to third-party applications. Among the affected services were Apple, Adobe Cloud, Docker's

Registry Hub, GitHub, GitLab, Quora, Slack, Trello, and Zendesk (Nichols, 2017; Novet, 2017). This incident highlighted the systematic challenges that can arise from a single point of failure in a critical infrastructure component in a highly interconnected environment. Such dependencies highlight the value of chaos engineering practices, which can simulate these kinds of failures in a controlled manner to identify potential vulnerabilities before they escalate into large-scale outages.

From a strategic standpoint, investing in Chaos Engineering can yield a competitive advantage. Organisations that demonstrate a commitment to reliability and resilience can differentiate themselves in the market. This is especially important for businesses in regulated industries, where system availability and data integrity are crucial. A comprehensive Chaos Engineering framework can serve as a testament to an organisation's commitment to quality and customer satisfaction.

In addition to the initial benefits, the return on investment (ROI) for chaos engineering becomes apparent over time. By proactively identifying and addressing vulnerabilities before they trigger costly outages, organisations can significantly reduce downtime, and the financial losses associated with it. This proactive approach not only results in short-term cost savings but also paves the way for long-term value creation. As illustrated above, in Figure 13, this value increases progressively as an organisation matures its Chaos Engineering practices (AWS, n.d.-a). The effectiveness of chaos experiments is increased by teams refining their processes and gaining a deeper understanding of system interdependencies. This leads to continuous improvements in system architecture and operational resilience.

7 The synergy between CI/CD and resilient applications

The principles and practices of Continuous Integration and Continuous Delivery (CI/CD) have transformed the software development industry by facilitating faster, more frequent, and more reliable releases. While the primary focus of CI/CD is often on speed and efficiency, its impact extends significantly to the resilience of the applications being developed and deployed. By integrating testing, automation, and feedback loops throughout the development lifecycle, CI/CD provides a robust foundation for building applications that can withstand failures and adapt to changing conditions. This chapter will explore the crucial relationship between CI/CD methodologies and the creation of resilient software systems, examining how specific CI/CD practices contribute to enhanced stability and fault tolerance.

Automation is integral to the effective management of CI/CD pipelines and the development of resilient applications. By automating repetitive and error-prone tasks throughout the software development lifecycle, organisations can significantly improve the speed, reliability and consistency of their releases, ultimately contributing to more stable and fault-tolerant systems (EPAM SolutionsHub, 2025). From code integration to deployment, and even infrastructure management, automation minimises risks associated with errors that human operators may commit (Saraev & Stormboard, 2023). Automated processes ensure that work proceeds according to predefined plans and that feedback is provided swiftly, all of which is essential for the development and maintenance of software with a high level of reliability (EPAM SolutionsHub, 2025).

A notable approach to automation that directly supports resilience is GitOps. GitOps is a declarative way to manage both infrastructure and application configurations using Git as the single source of truth (S. Singh, 2023). Any changes to the desired state of the system are made through Git pull requests, which are then automatically reconciled and applied to the target environment by automation tools. This approach provides several benefits for resilience. Firstly, it ensures that infrastructure and application deployments are version controlled and auditable, allowing easy rollback to previous stable states in the event of problems (S. Singh, 2023). Secondly, the declarative nature of GitOps fosters consistency and mitigates the risk of configuration drift, which can often lead to unanticipated system failures (Chen & Gerring, 2025). GitOps automates the reconciliation process, ensuring system integrity is maintained in the event of failures or manual interventions (Chen & Gerring, 2025).

Platforms such as GitHub Actions provide capabilities for automating CI/CD workflows. GitHub Actions enable developers to define custom workflows that are triggered by events within their GitHub repository, such as code pushes, pull requests, or scheduled events. Workflows can automate the entire build, test and deploy process, ensuring that every code change is automatically validated and deployed in a consistent manner (Merced, 2024).

From a resilience perspective, GitHub Actions supports running extensive test suites, including unit tests, integration tests, and even chaos engineering experiments in pre-production environments (Gaurav, 2023). Automated testing helps to identify potential weaknesses and failure points early in the development cycle. Additionally, GitHub Actions facilitate the automated deployment process across diverse environments with consistency, thereby minimising deployment risks. Its integration with other tools and platforms also allows for the automation of infrastructure provisioning and configuration management, which in turn contributes to the overall resilience of the application environment (GitHub, n.d.-a; Chen & Gerring, 2025).

In addition to GitOps and GitHub Actions, other aspects of automation within the CI/CD pipeline also contribute to resilience. Automated tools for infrastructure provisioning, such as Terraform, facilitate the creation of consistent and reproducible infrastructure, thereby minimising the likelihood of environment-specific issues (AWS, n.d.-f). Configuration management tools like Ansible and Chef ensure that servers and applications are configured correctly and consistently, minimizing configuration drift. Automated rollback scripts can be integrated into the deployment process to quickly revert to a previous stable version in case of a failed deployment (AWS, n.d.-f; Abstracta, 2024).

By reducing human error, ensuring consistency, providing rapid feedback and facilitating practices such as GitOps and the use of platforms like GitHub Actions, automation enables development teams to build and deploy software that is more stable, reliable and capable of withstanding failures (Abstracta, 2024).

8 Chaos engineering and the pipeline

Chaos Engineering, when integrated into the continuous integration/continuous delivery (CI/CD) pipeline, proactively identifies weaknesses in production systems and offers a powerful opportunity to build resilience into applications from the earliest stages of development (Sonaniya, 2025). By strategically incorporating controlled failure injection into the pipeline, teams can gain early insights into how their applications behave under adverse conditions, allowing them to address potential issues before they reach production and impact end-users (Sonar, 2024). This proactive approach involves the process of shifting resilience testing left within the development lifecycle, thereby fostering a culture of building robust and fault-tolerant systems from the get-go (Gremlin, n.d.). This chapter will explore the strategy and placement of chaos experiments within the pipeline, as well as the principals involved in designing effective experiments and the key considerations for automating failure injection.

8.1 Strategy and placement within the pipeline

The strategic placement of Chaos Engineering experiments within the CI/CD pipeline is vital to maximise their effectiveness and minimise potential risks. Different stages within the pipeline offer unique opportunities to inject failures and to validate resilience characteristics.

During the preliminary stages of the software development life cycle (SDLC), specifically in the integration testing phase, it is possible to incorporate lightweight chaos experiments. These experiments are used to verify the basic fault tolerance of individual components or microservices and their immediate dependencies (Grosch, 2023). These experiments might involve simulating network latency or injecting resource exhaustion into isolated environments. The objective at this stage is to identify fundamental resilience issues early in the development cycle, providing rapid feedback to developers.

As the code progresses to staging or pre-production environments, it becomes possible to conduct more comprehensive and realistic chaos experiments. These environments often mimic the production environment, allowing more complex failure scenarios to be simulated, such as database failures, network partitions or the failure of dependent services (Mukkara, 2023). The implementation of chaos experiments at this stage presents a valuable opportunity to validate the application's resilience under conditions that closely resemble real-world production incidents, without affecting actual users.

Automated experiments designed to test the resilience of these systems are initiated during deployment, ensuring that each new release undergoes resilience testing before being considered for production deployment.

Although the primary focus of pipeline integration is on pre-production environments, some organisations with mature Chaos Engineering practices may also consider running carefully controlled and scoped experiments in production environments as part of their continuous delivery process (Li, 2024). These experiments are usually carried out with robust safety mechanisms and monitoring in place to minimise any potential impact on users. A fundamental aspect of this process involves defining and limiting the *'blast radius'*, the potential scope and impact of an experiment. The aim is to ensure any failure remains contained and does not spread beyond the established boundaries. The purpose of these experiments is to gain real-world insight into the resilience of the application under actual production loads and conditions. It is necessary to carefully consider the placement and scope of such experiments and to have a high level of confidence in the overall resilience posture of the system.

The placement strategy for chaos experiments should match the organisation's risk tolerance and the maturity of its CI/CD and chaos engineering practices. A common approach is to start with simpler experiments in earlier stages and gradually increase the complexity and scope as confidence grows.

8.2 Designing effective experiments for the pipeline

To design effective chaos experiments within the CI/CD pipeline, it is essential to adopt a structured and scientific approach. This approach will ensure that the experiments yield meaningful insights into the application's resilience. The design process should be guided by several key principles.

Firstly, each experiment should start with a clear hypothesis about how the system should behave under a specific failure condition. This hypothesis should be based on an understanding of the system's architecture, dependencies, and expected resilience mechanisms (R., 2023). For example, a hypothesis might be: "If the database becomes temporarily unavailable, the application should gracefully degrade by displaying cached data and queuing new requests."

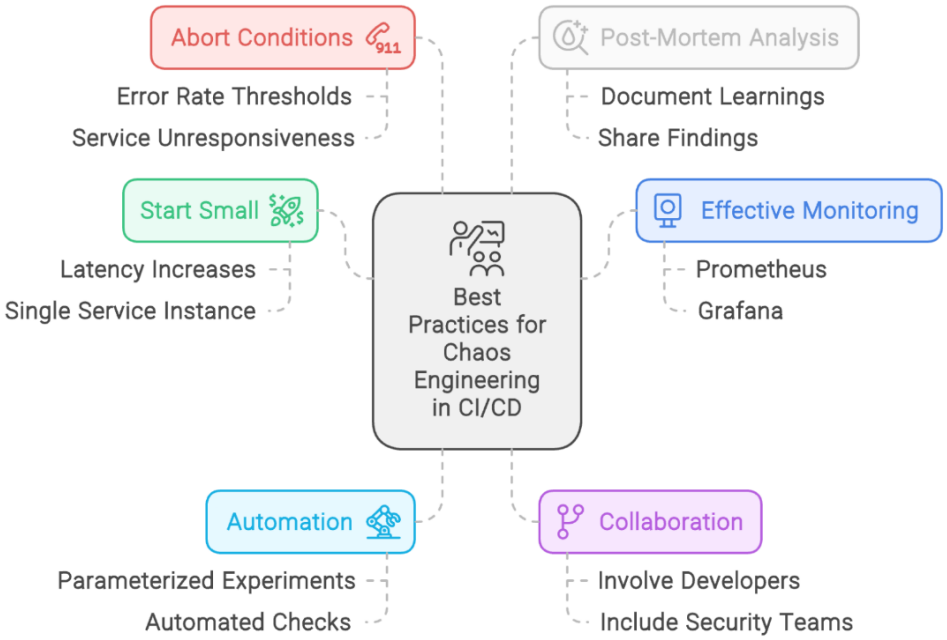
Secondly, it is very important to carefully define the scope of the experiment. This involves identifying the specific components or services that will be targeted for failure injection and

the duration of the experiment. Limiting the scope will help isolate the impact of the experiment and facilitate analysis of the results.

Thirdly, the type of failure injected should be relevant to the potential failure scenarios that the application might encounter in production. These could include network latency, packet loss, DNS failures, resource exhaustion (CPU, memory, disk), or the termination of specific instances or containers (Schillerstrom, 2022). The selection of failure type should be informed by the system's architecture and its critical dependencies.

Finally, it is essential to have clear metrics and observability in order to evaluate the outcome of the chaos experiment. Prior to conducting the experiment, baseline metrics for key performance indicators (KPIs) such as error rates, latency, and resource utilisation should be established. During and after the experiment, these metrics need to be closely monitored so that the application's behaviour can be checked against the initial hypothesis. Effective logging and tracing are also crucial for understanding the sequence of events and identifying any unexpected behaviour.

Figure 16. Best practices for chaos engineering in CI/CD Pipelines (Sonar, 2024).



As illustrated in Figure 16, a diagram with best practices for chaos engineering in CI/CD pipelines. The design of chaos experiments within the pipeline should be iterative. The results of each experiment should be analysed, and the findings should be used to refine the application's resilience mechanisms and inform the design of future experiments.

8.3 Practical aspects and considerations

The injection of failures within the CI/CD pipeline can offer significant benefits in terms of efficiency and consistency. However, it is essential to consider several key factors to ensure the safe and effective implementation of this automation.

The selection of the appropriate tools is a key consideration. There are various tools and frameworks available for automating failure injection, ranging from open-source tools like Chaos Mesh and Litmus Chaos to commercial platforms like Gremlin. The choice of tool will depend on the specific technologies used in the application stack and the desired types of failures to be injected (Gremlin, 2023a). Integration with the existing CI/CD platform is also an important factor.

When automating failure injection, it is also essential to implement safety mechanisms. Implementing safeguards to prevent unintended consequences and to minimise the potential impact of experiments is quite important (Grosch, 2023). This might include defining blast radii to limit the scope of failure injection, implementing automated abort conditions that can halt an experiment if critical metrics degrade beyond acceptable levels, and ensuring that all injected failures are reversible.

Comprehensive monitoring and alerting systems are crucial for conducting automated chaos experiments. Real-time visibility into system behaviour is essential for detecting any unplanned problems and validating the effectiveness of resilience mechanisms. Automated alerts should be configured to notify the relevant teams if any critical failures or significant performance degradations are detected.

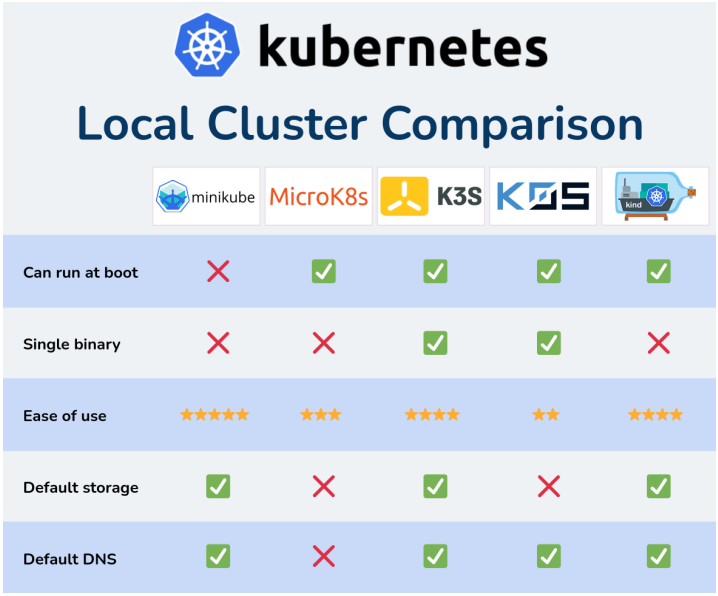
Lastly, the capability to seamlessly restore from injected failures is paramount. An automated process should ensure that the system can be rapidly restored to its usual operating state after the conclusion of an experiment, irrespective of the result. This might involve terminating injected faults, restarting affected services, or rolling back to a previous stable deployment (Grosch, 2023).

9 Implementation and setup

After exploring the theoretical aspects of microservices, CI/CD, and chaos engineering in the previous chapters, this chapter moves on to practical implementation. Chapter 9 describes the technologies used, the setup procedures, and the initial development of a microservices-based application. It also covers the creation of a CI/CD pipeline using GitHub Actions and the integration of basic chaos engineering. Additionally, a GitHub repository containing the application code is provided. The objective is to demonstrate how these modern practices can be translated into a functional and resilient system.

For the purposes of the practical portion of this thesis, K3s will be used. K3s is a lightweight Kubernetes distribution developed by Rancher (SUSE) and is often used for local development due to its ease of setup. As demonstrated in Figure 17, a comparison of popular Kubernetes distributions for local development is shown.

Figure 17. Comparison between local development Kubernetes distributions (Radwell, 2021).



K3s provides a straightforward and efficient way for establishing a cluster, offering a wide range of features out of the box (Lüders, 2025; Lunbeck, 2024). A key feature of K3s is the inclusion of Traefik as an integrated ingress solution.

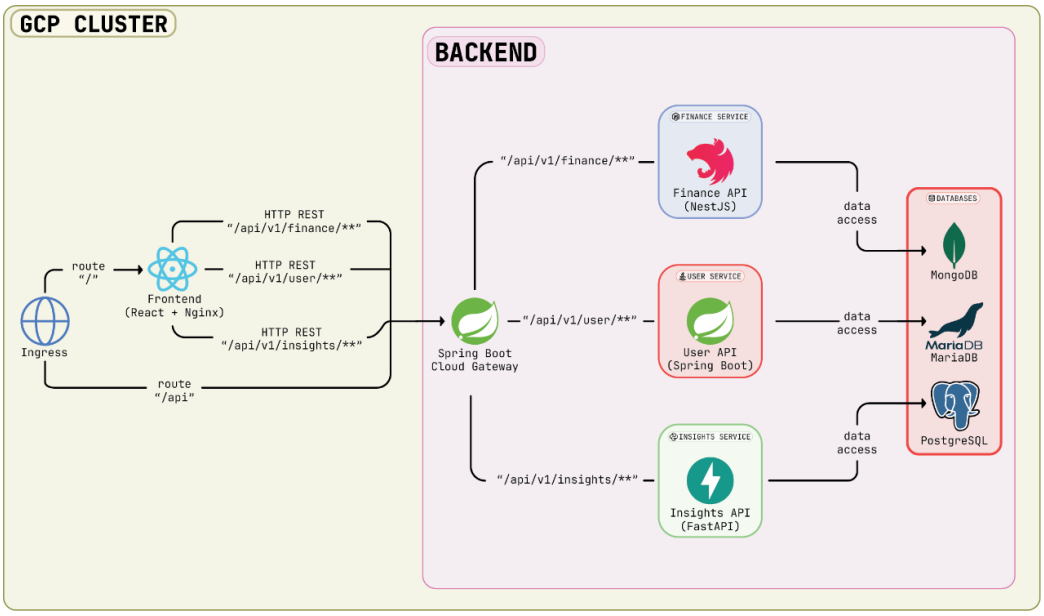
9.1 Microservices application development

For the purposes of practical testing, an application was developed. Wallet, a personal finance assistant designed to help users manage their budgets, transactions, bills, and financial goals, has been developed using a microservices architecture. The application comprises a front end service, a back-end gateway, and several core microservices, all communicating via HTTP RESTful APIs. The following subchapters will detail the architecture and technology stack of each component.

9.1.1 Overall architecture

The application is built following a microservices architecture, as shown in Figure 18, the system consists of three main layers: a front end service, a back end gateway, and several core back-end microservices.

Figure 18. General overview of Wallet application architecture (Bakema, 2025).



The user interacts with the application through the frontend service, which is built using React and served by Nginx. This layer is responsible for rendering the user interface and handling user interactions. All requests originating from the frontend are routed to the backend via the Backend Gateway, which is implemented using Spring Boot Cloud Gateway. The gateway functions as the sole entry point for all external requests, offering functionalities such as request routing, authentication and authorisation.

The core business logic components are located behind the gateway and have been implemented as individual back end microservices. These services are responsible for specific domains within the application. The User Microservice is responsible for user authentication and registration using Google OAuth2, accessible through the /api/v1/login endpoint. The Finance Microservice is responsible for managing the user's financial data, providing create, read, update and delete (CRUD) operations for bills, goals, transactions, and budgets via the /api/v1/finance endpoints. Lastly, the Insights microservice is responsible for analysing the financial data to generate financial reports and analytics, accessible via the /api/v1/insights and /api/v1/analytics endpoints.

The front end and back end gateway communicate using HTTP RESTful API requests, as do the back end microservices. This approach ensures a well-defined and standardised communication protocol throughout the application.

9.1.2 Front end microservice

The front end service of the Wallet application provides the user interface through which users interact with the personal finance assistant. The front end has been built using React, a JavaScript library for building interactive user interfaces and Single-Page Applications (SPAs), and served by Nginx, a high-performance web server. The front end is responsible for displaying the user's financial data and accepting their input to manage budgets, transactions, bills and goals.

The decision to use React was made based on its component-based architecture, which allows for the creation of reusable and maintainable UI elements. React's virtual DOM also facilitates efficient updates to the user interface, ensuring a smooth and responsive experience. The decision was also influenced by the fact that there was limited experience with the library. Working with React for this project has provided an additional learning experience. The front end service communicates with the backend of the application by making HTTP requests to the Spring Boot Cloud Gateway. All backend interactions utilise the /api/v1 prefix, ensuring a consistent API endpoint structure. React Query, a well-known React package, is utilised within the React application to manage these asynchronous API calls. State management within the React application, provided by Zustand, a state management package, ensures the data is efficiently managed and updated across different components and pages. Navigation between the application's different routes is handled by the React Router. The user interface (UI) has been implemented using Tailwind, a CSS library, and shadcn/ui, a component library. The deployment of the front end service involves the creation of a React application in the form of static files, which are

then served by the Nginx server. The deployment of the front end service involves building the React application into a set of static files, which are then served by Nginx. The build process is managed by Vite and Speedy Web Compiler (SWC), which are popular compilation and bundling tools in the JavaScript ecosystem.

Figure 19. Login screen of the application (own work).

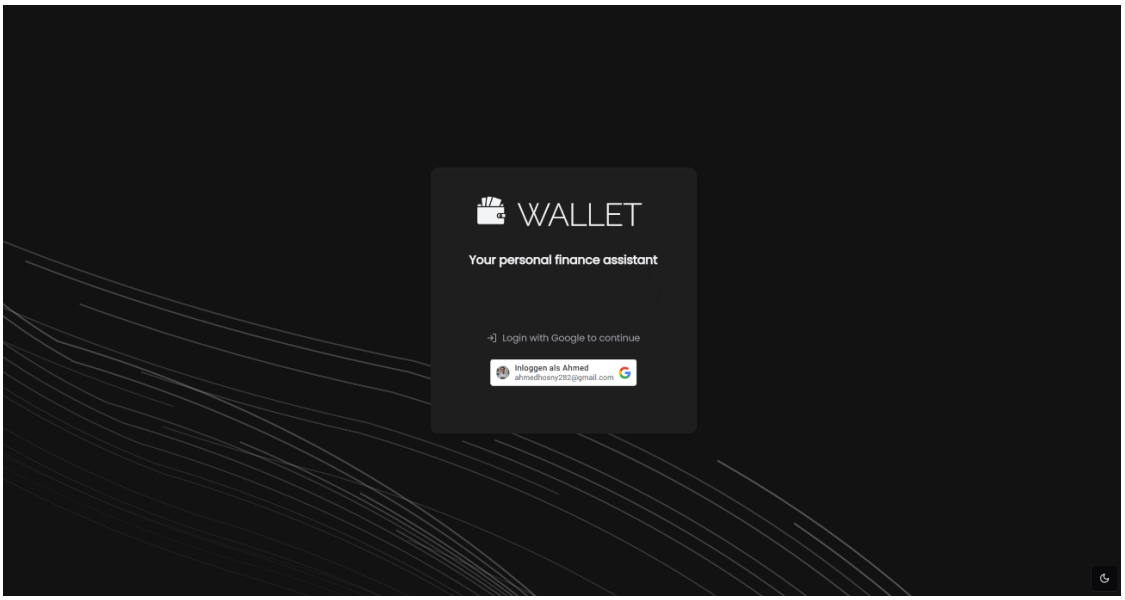
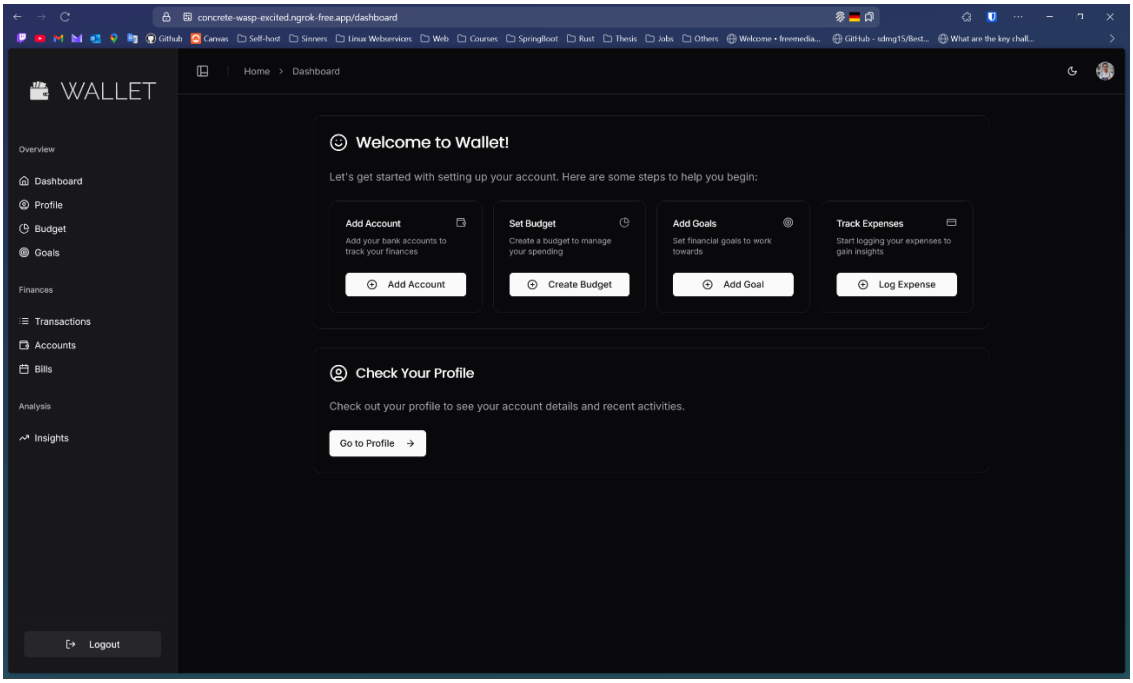


Figure 20. Onboarding screen of the application (own work).



As shown in Figure 19, the login screen of the Wallet application provides users with a simple and secure entry point to their account. Upon logging in, new users are greeted with

the onboarding screen (see Figure 20), which prompts them to add their data to initiate the process. This step ensures that users can personalise their experience and begin tracking their finances with ease.

9.1.3 Gateway microservice

The Back End Gateway service for the Wallet application has been implemented using Spring Boot Cloud Gateway. This service functions as the central entry point for all client requests originating from the front end service, providing a single interface to the various back end microservices. The selection of Spring Boot Cloud Gateway was motivated by its initial ease of setup as an API gateway specifically designed for microservices architectures, offering features such as dynamic route configuration, load balancing, and integration with other components of the Spring Cloud ecosystem.

A core function of the Back End Gateway is to route requests. It directs incoming HTTP requests to the appropriate back end microservice based on the request path. All requests intended for the backend are prefixed with /api/v1. The gateway is configured with routing rules that map these paths to the corresponding internal services. For example, a request to /api/v1/user/login is routed to the User microservice, while requests under /api/v1/finance are directed to the Finance microservice, and those under /api/v1/insights to the Insights microservice.

The gateway also serves as a reverse proxy, abstracting the internal architecture of the back end from the front end. The Front End Service only needs to be aware of the gateway's address, not the individual addresses or ports of each microservice. This provides a level of indirection that boosts security and allows changes to be made to the back end infrastructure without directly impacting the front end

The backend gateway can be configured to address additional overarching concerns, such as request logging and rate limiting, and thus protect backend services from potential misuse. It can also verify the origin of incoming requests, further enhancing security by restricting access to the back end. The rules for how requests are handled are normally set up using YAML or properties files, which makes it easy to change how requests are handled.

9.1.4 User microservice

The User microservice is responsible for managing user accounts within the Wallet application. It is built using Java Spring Boot and stores data in a MariaDB database. The service handles the creation and retrieval of user account information after the Backend Gateway has successfully checked the user's credentials.

The authentication process starts on the user's browser, using a React Google OAuth2 library. This allows users to sign in using their existing Google account. If this is successful, a request is sent to the User Microservice via the Backend Gateway.

The Backend Gateway plays a key role in authorising this request. It checks the JSON Web Token (JWT) provided by Google as part of the OAuth2 flow. If the JWT is valid, the Gateway forwards the request to the User Microservice. When the User microservice receives an authenticated request at its /api/v1/login endpoint, it checks the user's information (the email address and name obtained from the JWT). It then looks in the MariaDB database to see if this email address is already used by a user. Based on the information obtained the User microservice handles two scenarios: New user: If the email isn't found in the database, it creates a new user record, generating a new Universally Unique Identifier (UUID), and stores it along with relevant information from the JWT. Existing user: If there is already a user with the given email address, the User microservice updates the last_login timestamp for that user in the database. The User Microservice is made using Jakarta EE specifications and the Java Persistence API (JPA) for database interactions. It also uses several well-known design patterns to make sure the code is well-structured and easier to maintain, such as Dependency Injection (DI), Data Transfer Objects (DTOs) and the Repository Pattern. To make sure the User microservice is reliable, unit tests were put in during its development. These tests check that each part of the microservice works on its own. By adopting a 'shift-left' approach to testing, unit tests are written and executed early in the development lifecycle, which helped finding problems early on.

9.1.5 Finance microservice

The Finance Microservice is responsible for managing the main financial data within the Wallet application. This includes storing and retrieving information about users' financial goals, transactions, budgets, and recurring bills. This service was developed using NestJS, a Node.js framework built with TypeScript, and uses MongoDB as its main database.

NestJS was chosen for its well-defined approach to building scalable and maintainable server-side applications. Its design, based on Angular, uses modules, controllers and services, which helps to keep the code organised. NestJS was configured to use Fastify as its underlying HTTP framework, which was selected for its performance and efficiency in handling network requests. The choice of NestJS and Fastify was also made due to a desire to diversify the technology stack within the wallet application and gain experience with emerging JavaScript-based backend technologies.

MongoDB, a NoSQL document database, was chosen because it can handle different and changing financial data structures. It does not have a fixed structure, so it can easily be adapted to different types of financial records without needing complicated definitions. This flexibility is useful when developers are dealing with different types of information, like bills, goals, transactions and budgets. Using MongoDB also meant served as an opportunity to explore and manage a different database paradigm – NoSQL instead of SQL – within the project. For interaction with the MongoDB database, the Finance Microservice uses Mongoose, an object-relational mapper (ORM). This makes it easier to use MongoDB and checks that the database is correct. This helps to make sure that data is consistent and makes it easier to manage data in an object-oriented way. The Finance microservice exposes a set of RESTful API endpoints under the /api/v1/finance path to manage the user's financial data. These endpoints include: /finance/bill, /finance/budget, /finance/goal and /finance/transaction.

Each of these endpoints supports all CRUD operations, allowing users to create new financial records, retrieve existing ones, modify their details and delete them if desired. The service has controllers to handle incoming HTTP requests, services to contain the core business logic for managing financial data, and a data access layer that uses Mongoose to interact with the MongoDB database. To make sure the code is well-structured and easy to maintain, the Finance Microservice also uses Repository, Dependency Injection and DTO patterns. The Finance microservice mostly operates on its own data, but it can also use HTTP RESTful APIs to communicate with other microservices, like the Insights service, to send data for analysis.

9.1.6 Insights microservice

The Insights Microservice looks at the financial data that the Finance Microservice manages to create useful information for the user. This includes creating dashboards, providing analytics, making predictions about future spending, and detecting any unusual spending patterns or anomalies. This service was developed using FastAPI, a modern,

high-performance web framework for building APIs with Python, and uses PostgreSQL as its main database.

FastAPI was chosen for its speed and for the fact that it can automatically check data is valid, making it well suited to processing data and delivering analysis results. PostgreSQL was selected for its strength, reliability, and excellent support for complex queries, which are essential for data analysis.

The Insights microservice is a web service that can be found at /api/v1/insights/dashboard and /api/v1/insights/analytics. The /dashboard endpoint provides a summary of key financial metrics and visualisations, while /analytics offers more detailed reports and potentially allows users to request specific analyses.

To carry out its analysis, the Insights microservice retrieves financial data from the Finance microservice via HTTP RESTful API calls. This data, which includes transactions and budget information, is used to generate insights.

Currently, the service uses something called a 'random forest identifier' – an ensemble learning method for regression and classification – to predict future spending. While this method has its uses, it is known that it has its limits when it comes to making predictions about financial forecasts that include a time element. For this reason, a future enhancement is planned to implement Prophet, a library specifically designed for time series analysis. This will provide more accurate and meaningful predictions by looking at the user's financial data over time.

Internally, the FastAPI application consists of routes that handle the API endpoints, data models (defined using Pydantic), and logic for fetching data from the Finance service, performing the necessary analysis, and formatting the results for the API responses. The service interacts with the PostgreSQL database, which is used to store calculated insights, aggregated data and temporary results.

9.2 CI/CD pipeline implementation

To automate the software development lifecycle for the wallet application, a continuous integration and continuous delivery (CI/CD) pipeline was implemented using GitHub Actions. GitHub Actions was chosen for its tight integration with the project's version control setup, ease of use in defining automated workflows, and generous free tier for open source projects. The pipeline currently consists of three different workflows designed to automate

code quality checks, build and publish container images, and prepare for potential deployment to Google Cloud Platform (GCP).

9.2.1 Secrets management in GitHub Actions

An important aspect of setting up the CI/CD pipeline was the secure management of the sensitive configuration required for the wallet application. The application relies on environment variables for a variety of configuration purposes, including defining the addresses of other microservices, storing API credentials and tokens for external services, and specifying whether the application is running in a development or production environment. A .env.example file serves as a template for both local development and production, outlining the expected environment variables and their intended purpose as placeholders.

The actual values corresponding to these environment variables, especially those containing sensitive information, were securely stored as repository secrets for automated workflows within GitHub Actions. GitHub offers a secure way to manage these credentials within the repository settings (Settings -> Secrets and variables -> Actions -> Repository secrets). By configuring these values as secrets, the CI/CD workflows can access the necessary configuration without the risk of exposing sensitive information directly in the workflow configuration files or codebase, meeting security best practices. These secrets are then used by the workflows to configure the application during the build and deployment processes, to enable smooth interaction with other services, and to ensure that the correct environment settings are applied.

9.2.2 Workflow breakdown

The CI/CD pipeline for the Wallet application is made up of three separate workflows designed to automate different stages of the software delivery process.

Linting workflow: This workflow, defined in .github/workflows/lint.yml, is triggered on every push to the main branch and on every pull request. Its main purpose is to automatically check the codebase for stylistic and programmatic errors using different linting tools specific to each technology used in the application (e.g. ESLint for React and NestJS, linters for Java and Python).

The workflow involves checking out the code, setting up the required environment (Node.js, Java, Python), running the relevant linting commands, and reporting any issues found as

annotations on the pull request or as part of the output of the workflow run. A simplified example of the actual configuration used in the GitHub repository for this workflow is shown below.

```yaml
name: Lint

on: [push, pull_request]

jobs:
 lint:
  runs-on: ubuntu-latest
  steps:
  - uses: actions/checkout@v3
  - name: Set up Node.js
    uses: actions/setup-node@v3
    with:
      node-version: '21.7'
  - name: Install Frontend Dependencies
    run: cd frontend && npm install
  - name: Run Frontend Lint
    run: cd frontend && npm run lint
  # Further similar steps for other microservices
```

Docker Hub workflow: defined in .github/workflows/build-images.yaml, is triggered on each push to the main branch. It automates the building of Docker images for each microservice and pushing them to Docker Hub. The workflow includes checking out the code, then building each microservice using its respective Dockerfile located in each service's directory, logging into Docker Hub using the DOCKERHUB_USERNAME and DOCKERHUB_TOKEN secrets and finally pushing the tagged images to the specified Docker Hub repository. A simplified example of the actual configuration used in the GitHub repository for this workflow is shown below.

```yaml
name: Build and Push Docker Images
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]
jobs:
  build:
    runs-on: ubuntu-latest
```

```
env:
  DOCKER_HUB_USERNAME:  ${{secrets.DOCKER_HUB_USERNAME }}
  DOCKER_HUB_TOKEN: ${{ secrets.DOCKER_HUB_TOKEN }}
steps:
  - name: Checkout repository
    uses: actions/checkout@v4

  - name: Login to Docker Hub
    uses: docker/login-action@v3
    with:
      username: ${{ secrets.DOCKER_HUB_USERNAME }}
      password: ${{ secrets.DOCKER_HUB_TOKEN }}
      # Other environment variables

  - name: Set up Docker Buildx
    uses: docker/setup-buildx-action@v3

  - name: Build Docker images using Docker Compose
    run: docker compose build

  # Push all images at once to Docker Hub
  - name: Push images using Docker Compose
    run: docker compose push
```

A third workflow, named Deploy to GKE and defined in .github/workflows/gke.yml, was implemented with the intention of deploying the Wallet application to Google Kubernetes Engine (GKE). This workflow would trigger automatically upon successful completion of the Build and Push Docker Images workflow on the main branch. However, due to a lack of resources to fund cluster operations on GCP for testing purposes, this workflow remains untested at the present time. The intended steps of this workflow are as follows: The workflow starts by checking out the latest code from the repository. It then proceeds to set up the Google Cloud CLI, authenticating using the credentials stored in the GKE_SA_KEY GitHub secret and configuring the project ID from the GKE_PROJECT secret. The next step is to retrieve the necessary GKE credentials to interact with the specified cluster (GKE_CLUSTER secret) in the specified zone (GKE_ZONE secret). After retrieving the GKE credentials, the workflow installs the kubectl command line tool, which is used to interact with the Kubernetes cluster. It also configures Docker authentication with gcloud, allowing kubectl to pull images from the Google Container Registry (GCR) or Artifact Registry. The most important step is to apply the Kubernetes manifests located in the ./k8s/gcp directory of the repository.

This step will usually create or update the necessary Kubernetes resources defined in these files. This workflow outlines the automated steps for deploying the containerised Wallet application to a Google Kubernetes Engine cluster, following a successful image build and push. However, as mentioned above, this workflow has not been tested due to resource limitations. A shortened example of the configuration for this workflow is shown below.

```
name: Deploy to GKE

on:
 workflow_run:
  workflows: ["Build and Push Docker Images"]
  types: [completed]
  branches: [main]

jobs:
 deploy-to-gke:
  if: ${{ github.event.workflow_run.conclusion == 'success' }}
  runs-on: ubuntu-latest
  steps:
   - name: Checkout repository
     uses: actions/checkout@v4
   - name: Setup Google Cloud CLI
     uses: google-github-actions/setup-gcloud@v2
     with:
       service_account_key: ${{ secrets.GKE_SA_KEY }}
       project_id: ${{ secrets.GKE_PROJECT }}
   - name: Get GKE credentials
     uses: google-github-actions/get-gke-credentials@v2
     with:
       cluster_name: ${{ secrets.GKE_CLUSTER }}
       location: ${{ secrets.GKE_ZONE }}
   - name: Deploy to GKE
     run: kubectl apply -f ./k8s/
   - name: Force re-pull images
     run: |
       DEPLOYMENTS=$(kubectl get deployments -o jsonpath='{range
.items[]}{.metadata.name}{"\n"}{end}')
       for DEPLOYMENT in $DEPLOYMENTS; do
         kubectl rollout restart deployment/$DEPLOYMENT
       done
   - name: Verify deployment
     run: kubectl rollout status deployment/$DEPLOYMENT_NAME
```

9.3 Kubernetes local development environment setup

To test the application's microservices in an environment that closely resembled production, a local Kubernetes cluster was set up. This setup allowed for testing how the application works, trying out different Kubernetes settings, and making sure the microservices work well together on a container orchestration platform. For this, a lightweight Kubernetes distribution called K3s was used.

9.3.1 Vagrant virtual machine setup

Vagrant was the primary tool chosen to create and manage the local Kubernetes environment. Vagrant lets you define and create virtual machine environments that you can recreate again using a file called a Vagrantfile. A virtual machine running Ubuntu Server 22.04 was set up to create the Kubernetes cluster. VirtualBox is the platform used to host this virtual machine. This operating system was chosen because it is familiar, and because it is the most popular for running servers and cloud environments (Nazaryan, 2024). The specifications of the host device are not of significance as long as there is enough overhead.

The Vagrantfile was configured to allocate enough RAM (8 GB) and CPU (4 cores) to the virtual machine to run the Kubernetes cluster effectively. A shared folder was set up between the host system and the virtual machine, so that the YAML configuration files could be accessed easily from within the VM. This setup allowed the use of preferred development tools on the host machine while deploying and running the application within the virtualised environment.

The virtual machine was set up to connect to a private network and given a static IP address of 192.168.33.10. This IP address was the main way to access the Kubernetes cluster running inside the VM from the host machine. The choice of a private network was a deliberate one, to make it easier to potentially expand the cluster in the future by adding more nodes. This also made it easier to develop the system by keeping the necessary services separate and stopping unrelated network traffic from the standard network from getting in the way while debugging network issues.

For the full contents of the vagrant file used for this setup, see Appendix 2.

9.3.2 K3s installation and configuration

K3s, a lightweight distribution of Kubernetes created and maintained by Rancher, was selected due to its ease of installation and minimal resource footprint, making it well-suited for a local development environment. K3s was installed on the Ubuntu 22.04 virtual machine using the official installation script provided by the K3s team on their website. The installation was configured to create a single-node cluster, meaning that all the Kubernetes components (e.g. API server, scheduler, kubelet) run on the same virtual machine. This single-node setup served as a fully functional Kubernetes environment, suitable for local development and testing.

To interact with the K3s cluster from the host machine, the kubectl command-line tool was used. This meant setting kubectl up to work with the K3s API server, which was running in the virtual machine at the IP address 192.168.33.10. This was done by getting the kubeconfig file from the K3s installation in the VM and setting up the local kubectl to use this configuration.

Once the single-node K3s cluster was up and running within the virtual machine, the next step was to deploy the Wallet application's microservices. This was done using Kubernetes manifest files written in YAML. These files define how the application should be set up in the cluster. This includes how many copies (or replicas) of each microservice there should be, how they should be set up, and how they should be made available as services in the cluster. To be more specific, Kubernetes Deployment manifests were used to manage the microservices. This made sure that the right number of pod replicas were running and allowed for self-healing capabilities. Service manifests were used to show these microservices in the cluster and on the host machine via an Ingress. This allowed other microservices and the front end to access them.

These YAML files were used within the K3s cluster using kubectl with the command: kubectl apply -f <yaml-file>. After this, the deployment was verified by checking the status of the pods using kubectl get pods and attempting to access the services via their defined endpoints, either within the VM or from the host machine through port forwarding. An example of a Kubernetes Deployment manifest used for one of the Wallet application's microservices can be found in Appendix 3.

For the application's databases, MariaDB, PostgreSQL and MongoDB, it was essential to set up storage that would not be lost if the database containers were restarted or rescheduled. Kubernetes provides the concepts of Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) to manage persistent storage.

Persistent Volumes (PVs) are resources that represent a piece of storage. They are either allocated by an administrator or automatically allocated using Storage Classes. Persistent Volume Claims (PVCs) are requests for storage by users.

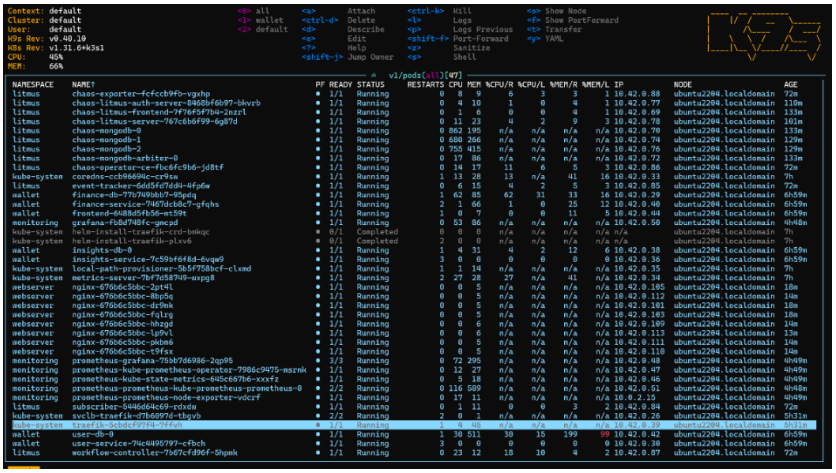
In this local K3s setup, the local storage provisioner that was included with K3s made it easy to create and manage persistent storage. This provisioner makes it easy to create and manage PVs (persistent volumes) based on the storage available on the virtual machine.

The deployment manifests for both the MariaDB and MongoDB microservices included Persistent Volume Claims. These PVCs specified how much storage capacity was needed and how the database pods could be accessed. When these were used with the K3s cluster, the PVCs triggered the creation of the Persistent Volumes. The local storage provisioner automatically created PVs on the underlying storage of the Vagrant VM. Kubernetes then linked – or bound, to use the correct technical term – the PVCs in the database pods to the newly created Persistent Volumes. This made sure that the database containers could access their own, permanent storage. Any data written to these volumes by the databases would be kept, even if the database containers were terminated or moved to a different node. In this single-node setup, they would remain on the same virtual machine.

9.4 Monitoring infrastructure setup

To get a full picture of how the Wallet application's microservices were running and performing within the local Kubernetes cluster, a robust monitoring infrastructure was established. This setup let us see the system's health in real-time, find and solve potential problems early, and learn how the system was using its resources. The primary tools utilised for monitoring the local Kubernetes environment were K9s, Helm, Prometheus, Grafana and Loki. To make sure that the monitoring components were organised and separated properly, a dedicated Kubernetes namespace called monitoring was created. This namespace contained all monitoring-related deployments, including Prometheus, Grafana and Loki, and separated them from the core application microservices running in the application’s main namespace. This approach aligns with the best ways of operating Kubernetes and makes managing observability tools in a microservices environment much easier.

Figure 21. K9s dashboard running on the cluster (own work)



K9s was the main tool for checking and managing the Kubernetes cluster. This user interface, which is terminal-based, provided a dynamic and real-time view of all Kubernetes resources, including pods, deployments, services and nodes. Figure 21 shows a snapshot of the K9s dashboard that was used during development. It provided quick inspection of resource status and logs and facilitated direct interaction with the cluster through kubectl commands executed within the K9s interface.

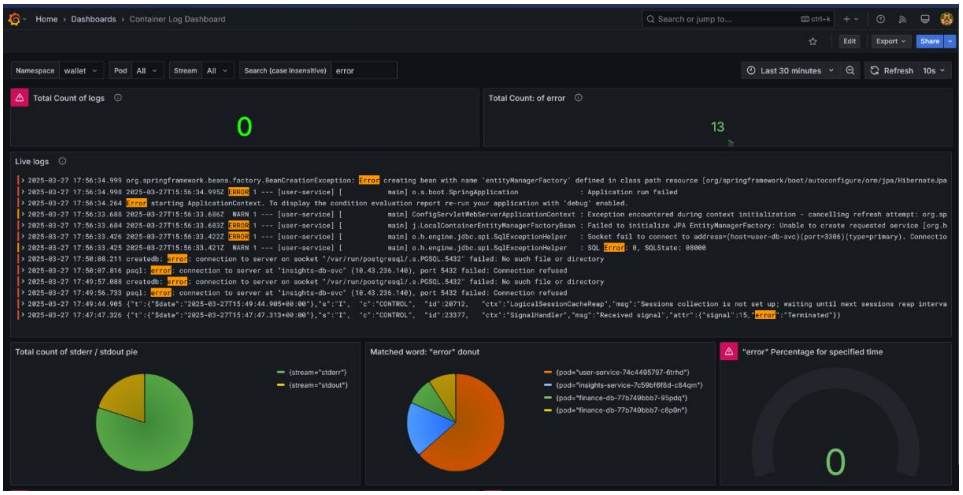
9.4.1 Deployment via Helm

Helm, a tool for managing packages related to Kubernetes, was used to make it easier to install and manage the monitoring tools. Helm charts, which are packages of pre-configured Kubernetes resources, were used to install Prometheus, Grafana, and Loki within the monitoring namespace. This made the setup process easier and made sure that these complex applications were set up with their recommended configurations. Helm, a tool for managing packages related to Kubernetes, was used to make it easier to install and manage the monitoring tools. Helm charts, which are packages of pre-configured Kubernetes resources, were used to install Prometheus, Grafana, and Loki within the monitoring namespace. This made the setup process much easier and made sure that these complex applications were set up with their recommended configurations. Using Helm charts made sure everything was set up the same way, reduced the chance of human error when installing things manually, and made it easier to update, roll back, or expand the monitoring stack when needed.

9.4.2 Monitoring Tools: Prometheus, Grafana, and Loki

Loki was deployed as a log aggregation system, providing a centralised and scalable solution for managing logs generated by the microservices. Similar to Prometheus, Loki focuses on indexing and grouping log streams based on labels, making it efficient for querying and analysing logs. Figure 22 illustrates a dashboard showing logs collected with Loki.

Figure 22. Aggregated logs collected by Loki (own work)



Prometheus was set up to collect and monitor core metrics. It is configured to collect metrics from various parts of the Kubernetes cluster, including the application microservices and the K3s cluster itself. It stores this data in a way that allows it to be analysed over time, and it can also create alerts based on rules that can been defined.

Grafana dashboards were created to show key metrics collected by Prometheus, such as how much CPU and memory pods were using, network traffic, request latency, and error rates. These dashboards were user-friendly interfaces for monitoring the overall health and performance of the wallet application, as well as the underlying Kubernetes infrastructure. Grafana was configured to connect to the Prometheus data source, enabling the creation of dynamic and informative dashboards as illustrated in Figure 23.

Figure 23. A dashboard created with Grafana (own work)

Within the monitoring infrastructure, Grafana dashboards were very important in assessing how well chaos experiments worked. This will be explained more in a later chapter of this work. These dashboards showed important information about the system in real-time, so that the impact of injected failures could be observed immediately.

By monitoring key metrics such as request latency, error rates, CPU and memory utilisation, the effectiveness of each chaos experiment in revealing potential vulnerabilities or demonstrating the resilience of the wallet application's microservices could be evaluated in real time.

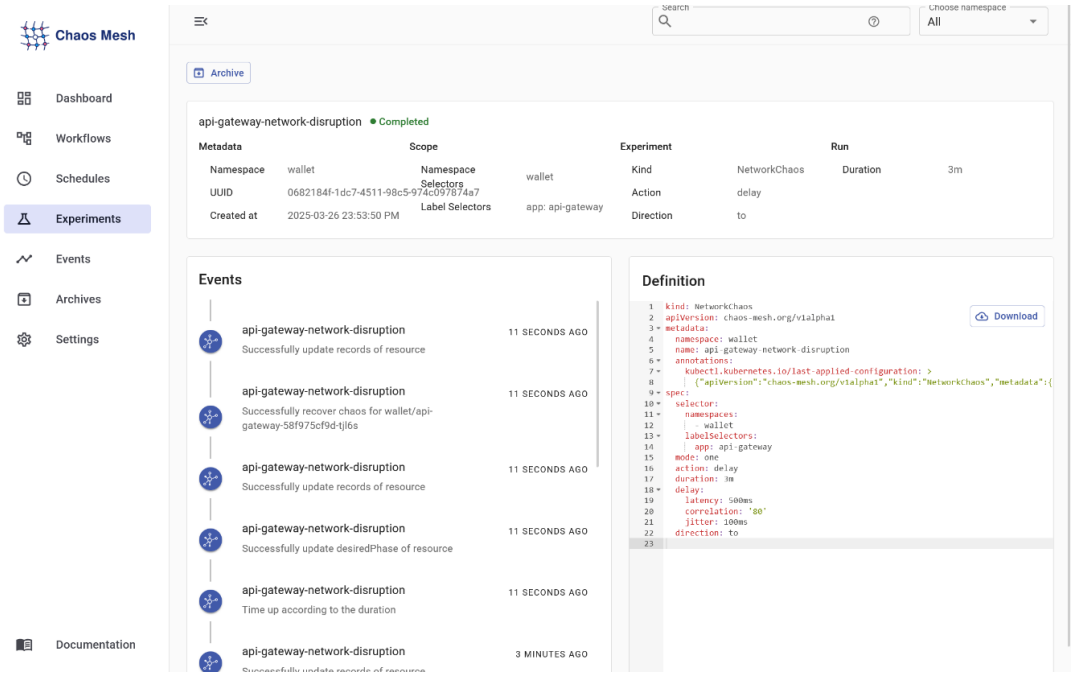
9.5 Chaos Mesh installation and configuration

Chaos Mesh, an open-source, cloud-native chaos engineering platform built on Kubernetes, was selected to simplify chaos engineering experiments and test the resilience of the Wallet application's microservices. This chapter provides details of the installation and configuration process of Chaos Mesh within the local K3s cluster.

The installation of Chaos Mesh was made easier using Helm, the Kubernetes package manager. The official Chaos Mesh Helm chart repository was added to the local Helm configuration, and the chaos-mesh chart was subsequently installed into a dedicated Kubernetes namespace named chaos-mesh. This made it easier to install all the necessary Chaos Mesh components, such as the chaos-controller-manager and chaos-daemon.

During the Helm installation, specific configuration parameters were set to tailor Chaos Mesh to the local K3s environment. For example, chaosDaemon.runtime and chaosDaemon.socketPath was configured to point to the containerd socket used by K3s /run/k3s/containerd/containerd.sock. This makes sure that Chaos Mesh can work with the containers managed by K3s. For consistency, a specific version of Chaos Mesh, 2.7.1, has been installed. In addition, metrics collection and ServiceMonitor were enabled to monitor Chaos Mesh itself, and the security mode of the dashboard was temporarily disabled for easier access in the local development environment. To allow Chaos Mesh to carry out different types of chaos experiments across the Kubernetes cluster, a ClusterRoleBinding – a special permission rule in Kubernetes – named chaos-mesh-full-access was created. This gave the chaos-mesh-controller-manager ServiceAccount in the chaos-mesh namespace full cluster-admin privileges. While this made it easier to experiment in the local environment, it is important to note that using such wide permissions might not be right for production use. The successful installation of Chaos Mesh was confirmed by checking the status of the pods within the chaos-mesh namespace with K9s. This confirmed that the Chaos Mesh controller and daemons were running as expected.

Figure 24. Details of an experiment shown on Chaos Dashboard (own work).



The availability of the Chaos Mesh dashboard was also confirmed by inspecting the services in the chaos-mesh namespace using kubectl get svc -n chaos-mesh chaos-dashboard. The Chaos Mesh dashboard offered a user-friendly web interface for defining, scheduling and monitoring chaos experiments, as shown in Figure 24. Access was gained by determining the NodePort assigned to the ChaosDashboard service and navigating in the web browser to http://localhost:<node_port>.

9.6 Automation using bash scripting

A bash script was developed to streamline the process of setting up the local development environment, including the Kubernetes cluster, monitoring tools and chaos engineering platform. The script was written to automate the installation and configuration of the various components, ensuring consistency and reducing the manual effort required to set up the environment.

The bash script is designed to guide the user through the installation of several key tools and configurations. It includes the following functionalities: displaying a visually appealing ASCII art header; verify prerequisites, such as the availability of sudo privileges; provide the option to install K3s; provide an optional installation of Chaos Mesh for conducting chaos engineering experiments; allow optional installation of Prometheus and Grafana for system monitoring and visualisation; allow for optional installation of Loki and Promtail for centralised log aggregation; provide installation of K9s, a terminal-based UI for Kubernetes cluster management; optional installation and execution of Ngrok to securely expose local services to the internet. The script asks the user which components they want to install using interactive prompts, so that they can choose based on their needs.

This Bash script made it easier to set up the local development environment in several ways. First, it reduced the number of steps that needed to be done manually by automating the installation of multiple tools and their configurations. This saved time and made sure everything was done in the same way each time. The script also offered flexibility by allowing users to only install the components they needed. In the end, the script made the whole process faster, more reliable, and easier to customise. For the complete Bash script, see Appendix 4.

10 Evaluation of resilience testing

This chapter evaluates the resilience of the Wallet application through a series of controlled chaos experiments. The goal was to proactively identify weaknesses and validate the application's ability to withstand various types of failures commonly encountered in distributed systems. Chaos Mesh, a Kubernetes-native chaos engineering platform, was employed to inject faults into the running application within the local K3s cluster.

The primary objectives of the resilience testing were to assess the application's fault tolerance capabilities under simulated failures, measure the recovery time of the application and its individual microservices, validate the effectiveness of implemented resilience patterns such as retries and potential circuit breakers, identify any critical single points of failure within the system, and gain a deeper understanding of the application's behaviour and performance under stress.

10.1 Methodology

A step-by-step plan for testing how well the system could deal with problems was used, and Chaos Mesh was used to create and carry out different tests to find weak points. A pre-defined Chaos Mesh Workflow, stress-reliability-test (detailed in Appendix 5), was used to automate a series of experiments. These experiments looked at different parts of the application's infrastructure. Key performance indicators (KPIs) were checked in real-time using Grafana dashboards, allowing for immediate observation of the system's response to the chaos.

Table 2. Experiments used within chaos workflow

Category	Experiment	Description	Expected Behaviour
Pod Failure	User Service Pod Failure	Simulated failure of the user-service pod	frontend shows an error message and recovers.
Pod Failure	Finance Service Pod Failure	Simulated failure of the finance-service pod	Graceful degradation and recovery; minimal frontend impact.
Network Disruption	API Gateway Network Chaos (Delay)	Introduced 500ms delay with jitter to the api-gateway	Increased latency in API responses.
Network Disruption	Inter-Service Network Partition	Simulated network partition between key microservices	Error messages; recovery after experiment ends.
Resource Constraint	User Database Memory Stress	Injected memory stress on the user-db	Higher latency and degraded performance for user service.
Resource Constraint	Finance DB CPU Stress	Injected CPU stress on the finance-db	Higher latency and degraded performance for finance service.

The workflow of the stress-reliability-test included experiments grouped into categories such as pod failures, network disruptions, and resource constraints. Table 2gives an overview of these experiments.

11 Discussion of results

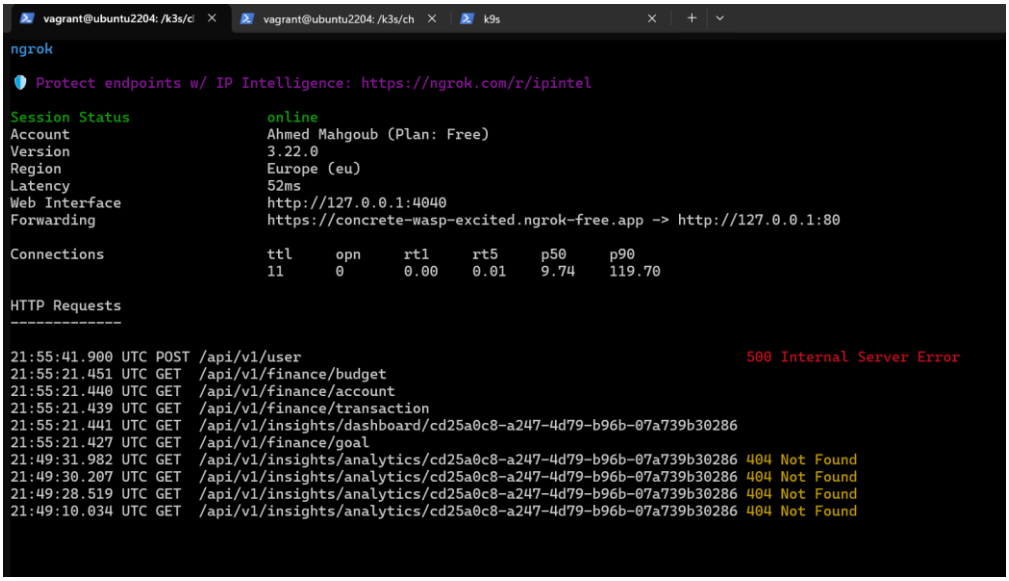
This chapter talks about the results and conclusions from the chaos experiments described in chapter 10. The aim of the experiments was to see how well the application's different microservices could deal with different types of fail scenarios.

11.1 Observed bugs and efficacy of experiments

The chaos engineering experiments carried out during the evaluation phase – often complemented by manual testing – revealed several critical and non-critical failures that would likely not have been detected by conventional unit or integration testing. These failures occurred primarily during scenarios involving induced network latency and intentional resource depletion, which provided valuable insight into the resilience and fault tolerance of the system under unfavourable conditions.

In scenarios where artificial network delays were introduced, one of the most notable errors was observed in the login process. The system failed authentication attempts inconsistently, resulting in failed user logins, especially when the user retried the login action multiple times during periods of high latency. These problems showed that the system did not handle timeouts and retries well, especially when the network was unreliable. The frontend was eventually exposed externally using Ngrok. When there was a network delay, the Ngrok terminal logged a "500 Internal Server Error" message, which was linked to an API Gateway timeout on login requests. This showed that the backend wasn't able to handle delayed login requests effectively, as illustrated in Figure 25.

Figure 25. Ngrok dashboard showing log in error (own work).



A more serious issue was found when attempting to log in simultaneously from two different browser tabs or after refreshing and attempting to log in again under delayed network conditions. Due to a combination of a race condition and insufficient safeguards for concurrent authentication requests, the system duplicated the same user by creating two separate entries in the database. The creation of this duplicate account exposed a significant flaw in the idempotency and concurrency control of the user service's account creation logic. From a database perspective, this incident highlights a failure to maintain ACID (Atomicity, Consistency, Isolation, Durability) principles, particularly the isolation aspect, which should prevent concurrent transactions from interfering with each other in a way that leads to data inconsistencies. It also highlighted a lack of transactional consistency in the account creation logic, which needs to be addressed to prevent data corruption in production environments.

In a separate experiment focused on inducing timeouts, various services were unable to complete inter-service API calls within the allowed time windows. This resulted in degraded functionality across the application, with certain features becoming temporarily unavailable. These issues were indicative of improper fallback mechanisms and lack of circuit breaker patterns to gracefully handle back-end service unavailability.

The stress tests looked at how well services worked when the system was under a lot of pressure. These tests made sure that some of the system's resources were used up on specific nodes. In these conditions, services like the front end (which was written in NestJS) and the PostgreSQL database performed in a stable manner and remained operational. However, the Java-based user service and API gateway failed to cope with the demand, resulting in a crash due to out-of-memory (OOM) and CPU depletion experiments.

Kubernetes' self-healing capabilities reacted as expected, detecting the failed pods, terminating them and spinning up new ones. During this recovery period, Kubernetes also rerouted incoming traffic to available pods, ensuring partial continuity of service and avoiding degraded functionality.

This behaviour was clearly observed in monitoring dashboards configured via Grafana. Real-time visualisation of high CPU and memory usage patterns during stress tests, alongside tracking of pod crashes and restarts using K9s, provided further insights into system performance. Figure 26 shows the CPU and memory spikes at the peak of the experiment.

Figure 26. CPU and memory utilisation during the experiment (own work).

Another screenshot (see Figure 27 and Figure 28) from K9s illustrates the crash and subsequent recreation of the user service pod.

Figure 27. Pods crashing due to OOM scenario (own work).

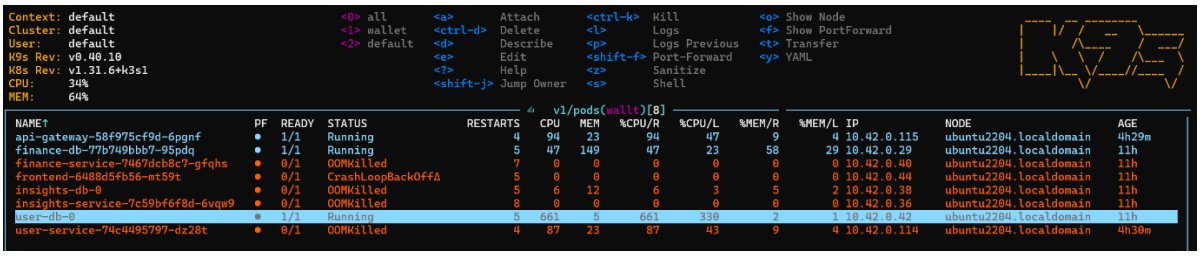
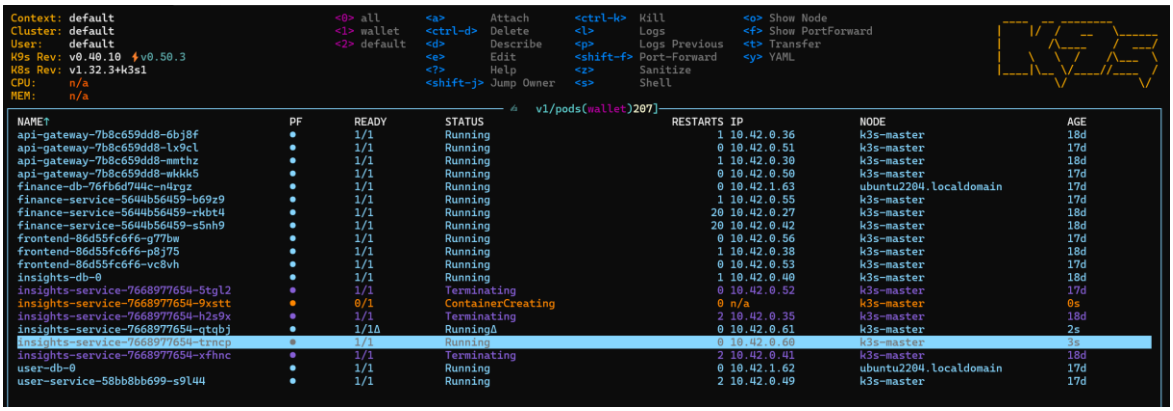


Figure 28. Kubernetes self-healing capabilities recreate crashed pods (own work).



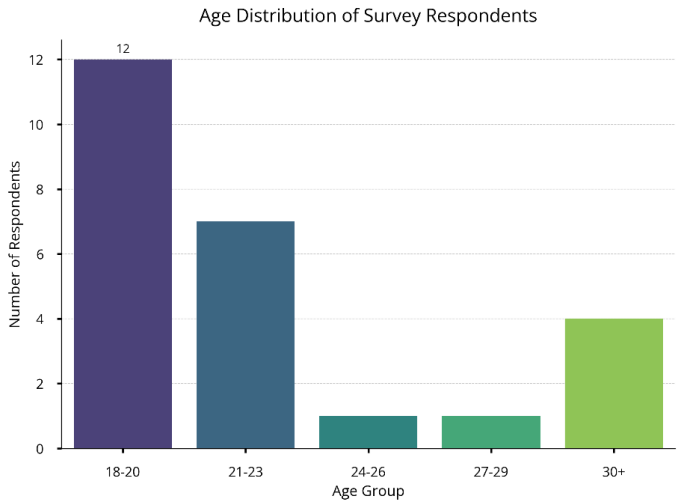
Although Kubernetes has been shown to provide solid self-healing and traffic routing capabilities, the experiments confirmed that the application stack itself needs to be improved to withstand failures more gracefully. Specifically, the Java-based services would benefit from stricter resource limit enforcement and potential runtime optimisations. By

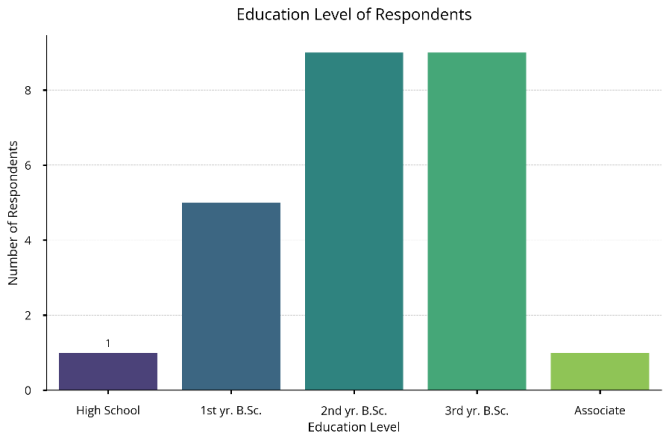
setting tight but realistic limits in the deployments, these services can be better isolated and prevented from excessive use of server resources and thus prevent system-wide instability. Additionally, it is recommended that resilience patterns such as retries, timeouts, and circuit breakers be considered across all services to improve fault isolation and end-user experience during partial outages. Overall, the experiments achieved their objective of uncovering hidden system vulnerabilities and validating the resilience strategy under test. They delivered usable insights into resource management, service behaviour under stress, and the effectiveness of Kubernetes as an orchestration and recovery platform. The combination of automated chaos engineering tools, observability platforms, and manual intervention provided a comprehensive evaluation of the system's ability to detect, absorb, and recover from failure.

11.2 Analysis of questionnaire feedback

A questionnaire was distributed among IT students and professionals to gain insight into the IT community's awareness of and perceptions of chaos engineering. The aim of the questionnaire was to gain insight into their familiarity with the concept, their views on its potential benefits and drawbacks, and their interest in learning more. A total of 25 responses were collected, offering a rich perspective from individuals with a variety of backgrounds in the field. An analysis of the demographic data illustrated in Figure 29 and Figure 30 indicates that the majority of respondents were students aged between 18 and 23, primarily pursuing bachelor’s degrees.

Figure 29. Age distribution of survey respondents (own work).

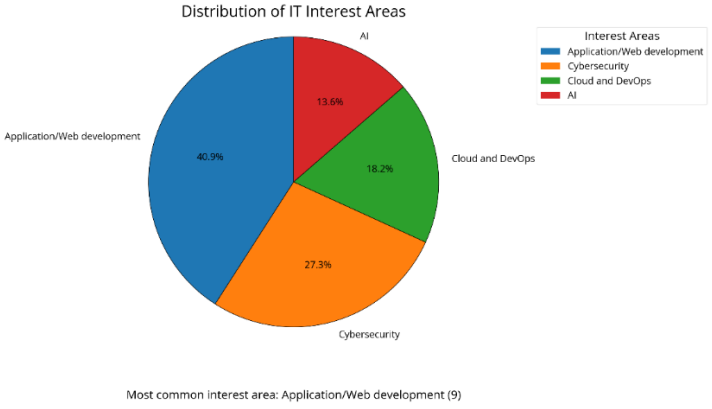




wn work).

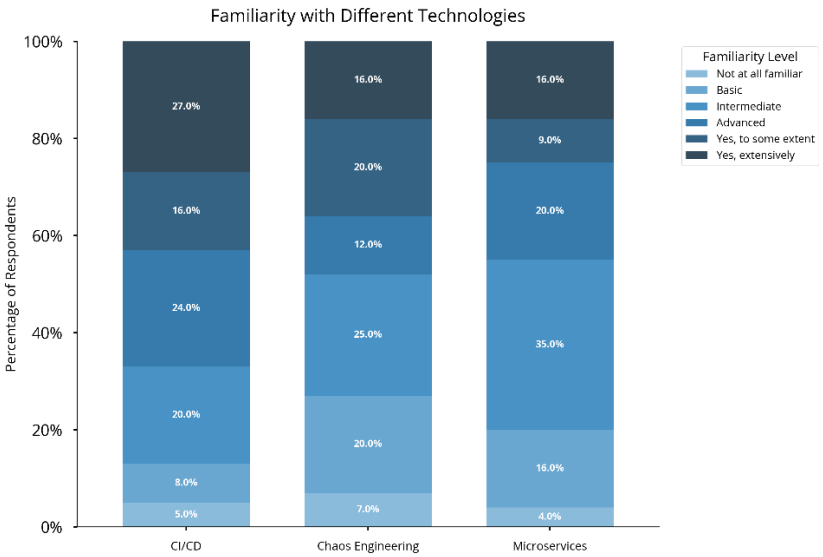
This suggests a strong representation from individuals who are either entering or early in their careers in IT. The wide range of areas of interest within IT, as demonstrated in Figure 31, extends from application development and cybersecurity to cloud and DevOps, reflecting a diverse spectrum of perspectives on software development and operations.

Figure 31. Distribution of IT field interest (own work).



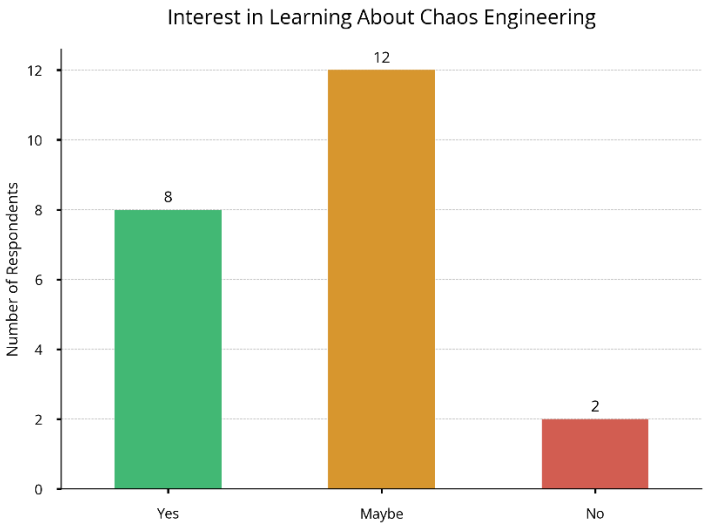
In terms of familiarity with chaos engineering, Figure 32 shows that a significant proportion of respondents said they were 'not at all familiar' with the concept. While some had a basic or intermediate understanding, this suggests that chaos engineering is still a relatively novel topic for many in the IT community, particularly at the student level.

Figure 32. Familiarity with different technologies (own work).



Despite this lack of widespread familiarity, Figure 33 shows an interest among respondents in learning more about chaos engineering, indicating a potential eagerness for knowledge in this area.

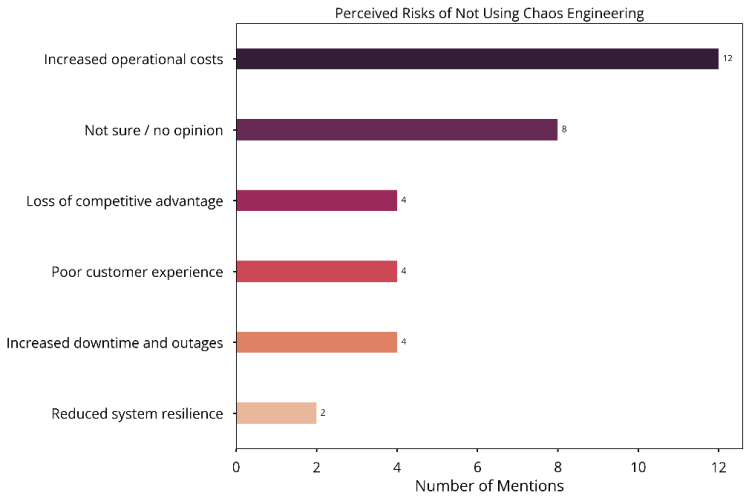
Figure 33. Interest in learning more about chaos engineering (own work).



This suggests a promising level of curiosity about the concept, suggesting that with proper education and well-organised exposure, chaos engineering could gain traction in the IT community, especially among students, as a proactive approach to system resilience. When asked about the potential risks of organisations not adopting chaos engineering practices, the responses, visualised in Figure 34, commonly pointed to concerns about reduced system resilience and increased downtime or outages. This is consistent with

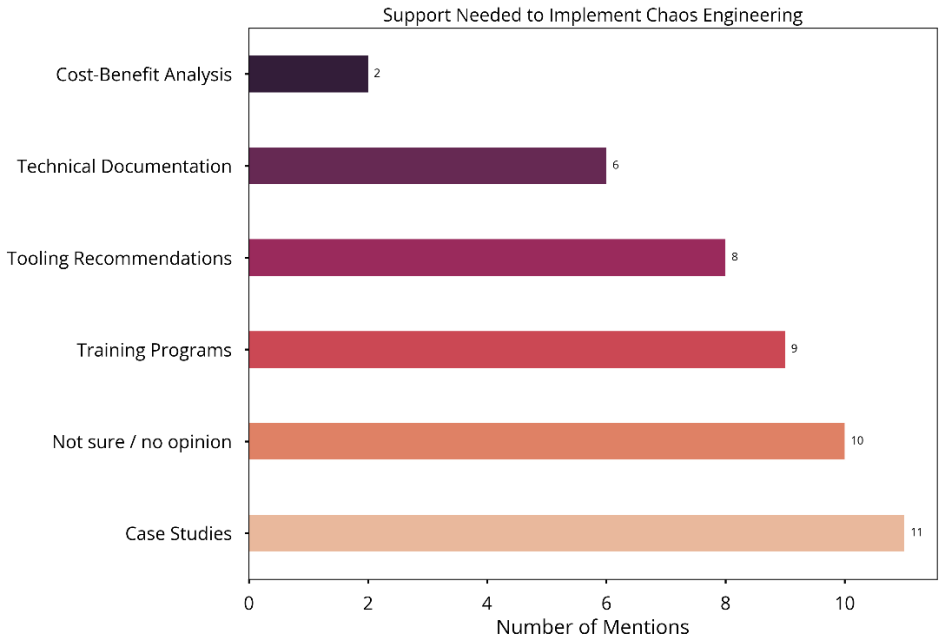
chaos engineering's core value proposition of proactively identifying and mitigating system vulnerabilities.

Figure 34. Perceived risks of not using chaos engineering (own work).



In addition, when asked what additional information or support would help them better understand or consider implementing chaos engineering, Figure 35 shows a strong preference for case studies and real-world examples. This shows that it is important to have practical examples and clear evidence of the benefits of chaos engineering for wider adoption and understanding.

Figure 35. Potential enablers of chaos engineering adoption (own work).



To sum things up, the answers to the questionnaire show that although chaos engineering might not be a well-known idea among most people in the IT community right now, there is an understanding of how it could help deal with important problems like system downtime and resilience. The strong interest in learning more, combined with the perceived risks of inaction, suggests a growing awareness of the importance of proactive resilience strategies.

Considering the increasing importance of resilient and reliable software systems and the apparent interest of students in learning more about this field, universities should also consider incorporating chaos engineering education into their IT and computer science curricula to better prepare future specialists with the necessary skills and knowledge. This feedback underscores the importance of the resilience testing conducted on the application, offering a practical example to help understand and adopt the principles of chaos engineering.

For a detailed breakdown of the responses to each question, please refer to Appendix 5.

12 Conclusion

The process of developing and evaluating the application as a microservices system on Kubernetes has highlighted key aspects of building resilient, reliable software in today's complex landscape. This research successfully addressed its core questions: practical development and deployment clarified the transition from monoliths, and the role of Kubernetes, while chaos engineering was crucial to proactively uncover and identify vulnerabilities in the system, with further Insights into business benefits and risk management gained from evaluation and questionnaire data.

The application of chaos engineering, through the deliberate injection of failures, served as a powerful lens into the application's inner workings under duress. The experiments revealed that while Kubernetes offers a vital layer of self-healing, the inherent complexities of microservices introduce new challenges. The discovery of a concurrency-related bug that leads to potential duplication of user accounts during network latency highlights a fundamental issue in distributed systems: maintaining data integrity in the face of asynchronous communication and potential delays. This finding emphasises the critical need for robust idempotency mechanisms and transactional consistency in microservice design, especially when handling sensitive operations such as user creation.

Additionally, the differential resilience observed across services during resource depletion experiments (e.g. Java services proving more vulnerable than NestJS) underscores the importance of technology-specific resource management and runtime optimisations. This suggesting tailored, rather than universal, strategies are necessary for each service.

The results of the questionnaire indicate an emerging awareness of chaos engineering within the IT community. The expressed interest in learning more, combined with a recognition of the risks associated with neglecting proactive resilience measures, signals a positive shift in perception of the importance of this discipline.

This research demonstrates that building resilient microservices is an iterative process requiring a blend of architectural best practices, sound testing methodologies, and a culture of embracing failure as a learning opportunity. As outlined in this thesis, the application of chaos engineering provides a solid foundation for achieving this objective, offering actionable insights into system behaviour under stress and paves the way for the development of more fault-tolerant applications.

13 Future work

Following on from the results and experience gained from this project, there are a number of areas that could be explored in future work. These could include ways to make the Wallet application more resilient and to increase understanding of chaos engineering principles.

A key area of future work is to expand the scope and complexity of the chaos experiments. This could include designing and running more complex scenarios that combine multiple types of failures simultaneously, such as inducing network latency while simultaneously taxing CPU resources. Running experiments for longer durations might also reveal longer-term stability issues or memory leaks that might not be apparent in shorter tests. Deploying the application in a more production-like environment, such as a multi-node cluster or cloud platform, and running chaos experiments there would provide valuable insights into its resilience at scale and in different infrastructure configurations.

Another important avenue for future work is to implement and evaluate specific resiliency patterns within the wallet application. For example, implementing circuit breaker patterns in the API gateway and user service could prevent cascading failures and improve the application's ability to gracefully handle unavailable downstream services.

Introducing rate limiting on the API Gateway could protect backend services from being overwhelmed by excessive requests, especially during transient failures. Introducing exponential backoff and jitter to enhance the existing retry mechanisms could improve the success rate of transient failures without overwhelming the failing service. It is also worth noting that future work could focus on implementing a solid solution to the duplicate user creation bug, by ensuring that the user creation endpoint in the user service is truly idempotent, to adhere to ACID principles.

Another interesting area for future work is to improve the monitoring and observability of the application. While current tools provide basic measurements and visualisation. Adding extra platforms that can observe, and measure could provide more detailed insights into how applications behave, especially when they are put under stress during chaos experiments. For Java-based services, using Spring Boot Actuator can be particularly helpful. It shows a lot of production-ready endpoints that provide detailed health checks, metrics, environment variables and more - all of which are essential for diagnosing faults and understanding the state of the system during fault injection scenarios.

Future research could also have an emphasis on user experience during failure scenarios. Experiments that focus on the direct impact on the user experience could provide important insights, such as tracking error rates, perceived latency, and user task completion during failure scenarios.

To gain a better understanding of the IT community's views on chaos engineering, further questionnaire research could be conducted with a larger and more diverse group of IT professionals and students. The formulation of more specific questions based on the results of the initial questionnaire could dig deeper into certain aspects of the understanding of chaos engineering and the barriers to its adoption.

Finally, it would be a big step towards automating resilience testing as an important part of the software development lifecycle to explore by adding chaos experiments into the continuous integration and continuous delivery (CI/CD) pipelines.

14 Author’s reflection

Writing this thesis has been an incredibly educational and insightful experience. It has provided an opportunity to explore the world of microservices, which lead to improving understanding of the complexities and possibilities within modern application design. This project also offered a great opportunity to work with technologies that were previously unfamiliar, such as NestJS for the backend. Navigating the intricacies of these tools presented a steep yet rewarding learning curve.

Throughout this process, moments of frustration were encountered. The frontend development process involved working with more advanced React concepts and its associated libraries, which presented certain challenges. Managing environment secrets across different deployment stages also presented its own set of pitfalls. Additionally, the need to move between Linux and Windows environments during development introduced platform-related issues that required a fair amount of troubleshooting and adaptation.

Despite the challenges encountered, writing this thesis has broadened the perspective on software development. Concepts previously unfamiliar, most notably the field of chaos engineering, were introduced through the research.

Overall, the process has contributed meaningfully to professional development in software engineering with successes and setbacks. While further learning remains necessary, the research conducted has served as a significant step forward in acquiring knowledge, skills and competence in these dynamic and ever-changing fields.

References

Abstracta. (2024, December 6). *DevOps automation explained: Boost efficiency and quality!* https://abstracta.us/blog/devops/devops-automation-explained-boost-efficiency-and-quality/

Anand, S. (2025, February 5). How Kubernetes ensures resilience during failures. *DevOpsNow*. https://devopsnow.hashnode.dev/how-kubernetes-ensures-resilience-during-failures

Arcot, K., & Vitucci, C. (n.d.). How relevant is chaos engineering today? *Qentelli*. Retrieved April 6, 2025, from https://qentelli.com/thought-leadership/insights/how-relevant-is-chaos-engineering-today

Atlassian. (n.d.-a). 5 advantages of microservices [+ disadvantages]. Retrieved April 3, 2025, from https://www.atlassian.com/microservices/cloud-computing/advantages-of-microservices

Atlassian. (n.d.-b). Microservices security: How to protect your architecture. Retrieved April 3, 2025, from https://www.atlassian.com/microservices/cloud-computing/microservices-security

Atlassian. (n.d.-c). Microservices vs. monolithic architecture. Retrieved April 3, 2025, from https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

Atlassian, & Pittet, S. (n.d.). Continuous integration vs. delivery vs. deployment. Retrieved April 5, 2025, from https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

Awad, J. W. (2025, March 23). Microservices pattern: Distributed transactions (SAGA). *Medium*. https://medium.com/@joudwawad/microservices-pattern-distributed-transactions-saga-92b5e933cea1

AWS. (n.d.-a). Downtime costs and the emergence of chaos engineering - AWS prescriptive guidance. Retrieved April 6, 2025, from https://docs.aws.amazon.com/prescriptive-guidance/latest/strategy-chaos-engineering-journey/chaos-engineering-emergence.html

AWS. (n.d.-b). Monolithic vs microservices - Difference between software development architectures. Retrieved April 3, 2025, from https://aws.amazon.com/compare/the-difference-between-monolithic-and-microservices-architecture/

AWS. (n.d.-c). SOA vs microservices - Difference between architectural styles. Retrieved April 4, 2025, from https://aws.amazon.com/compare/the-difference-between-soa-microservices

AWS. (n.d.-d). What are microservices? Retrieved April 2, 2025, from https://aws.amazon.com/microservices/

AWS. (n.d.-e). What is cloud native? - Cloud native applications explained. Retrieved April 2, 2025, from https://aws.amazon.com/what-is/cloud-native/

AWS. (n.d.-f). What is infrastructure as code? - IaC explained. Retrieved April 8, 2025, from https://aws.amazon.com/what-is/iac/

Bakema, D. (2025, April 12). Wallet application architecture.

Bhardwaj, A. (2024, May 13). How to use GraphQL to build backend-for-frontends (BFFs). *Hygraph*. https://hygraph.com/blog/graphql-backend-for-frontend

Bhattacharjee, S. (2024, October 10). Microservices testing: Strategies, tools, and best practices. *vFunction*. https://vfunction.com/blog/microservices-testing/

Bhattacharjee, S. (2025, March 10). Microservices architecture and design: A complete overview. *vFunction*. https://vfunction.com/blog/microservices-architecture-guide/

Chaos Mesh. (n.d.). Chaos mesh overview. Retrieved April 6, 2025, from https://chaos-mesh.org/docs/

Chen, B., & Gerring, S. (2025, January 15). Understanding GitOps: Key principles and components for Kubernetes environments. *Datadog*. https://www.datadoghq.com/blog/gitops-principles-and-components/

Cloud Security Alliance [CSA]. (n.d.). Microservices + cybersecurity. Retrieved April 5, 2025, from https://cloudsecurityalliance.org/research/topics/containerization

Corp, I. (2024a, September 3). ITIC 2024 hourly cost of downtime report part 1. https://itic-corp.com/itic-2024-hourly-cost-of-downtime-report/

Corp, I. (2024b, September 10). ITIC 2024 hourly cost of downtime part 2. https://itic-corp.com/itic-2024-hourly-cost-of-downtime-part-2/

Das, A. (2024, July 30). How to use version control in CI/CD pipelines. *PixelFreeStudio Blog*. https://blog.pixelfreestudio.com/how-to-use-version-control-in-ci-cd-pipelines/

Dass, V. (2024, September 23). Scaling applications with Kubernetes: Best practices. *BuildPiper*. https://www.buildpiper.io/scaling-applications-with-kubernetes-best-practices/

Darktrace. (n.d.). Anomaly detection | Definition & security solutions. Retrieved April 6, 2025, from https://www.darktrace.com/cyber-ai-glossary/anomaly-detection

Datadog. (2022, March 9). Audit logging: What it is & how it works. https://www.datadoghq.com/knowledge-center/audit-logging/

Design Gurus. (n.d.). How do you manage data consistency in a microservices architecture? Retrieved April 5, 2025, from https://www.designgurus.io/answers/detail/how-do-you-manage-data-consistency-in-a-microservices-architecture

Dwyer, J. (2017, December 17). The importance of CICD testing & how to test effectively. *Zeet*. https://zeet.co/blog/cicd-testing

Dwyer, J. (2023, November 11). What is Kubernetes continuous deployment (CD) & CI/CD best practices. *Zeet*. https://zeet.co/blog/kubernetes-continuous-deployment

Elxecoraline. (2024, February 22). Scaling agility: How microservices empower rapid development and deployment. *Medium*. https://medium.com/@elxecoraline/scaling-agility-how-microservices-empower-rapid-development-and-deployment-1e2cd058dbdb

EPAM SolutionsHub. (2025, January 27). Benefits of test automation. https://solutionshub.epam.com/blog/post/benefits-of-test-automation

Eyevinn Technology. (2020, February 19). Example of microservices for a streaming service. *Medium*. https://eyevinntechnology.medium.com/example-of-microservices-for-a-streaming-service-e2b2c556442e

Fahmy, M. (2024, November 15). Microservices architecture: Benefits, challenges, and use cases. *Medium*. https://miladezzat.medium.com/microservices-architecture-benefits-challenges-and-use-cases-9cec05adcb57

Gaur, N. (2021, September 10). Why Kubernetes is de-facto choice for every company going cloud native. *LinkedIn*. https://www.linkedin.com/pulse/why-kubernetes-de-facto-choice-every-company-going-cloud-nitin-gaur/

Gaurav, U. (2023, June 30). GitHub chaos actions in your CI/CD workflow [Part-1]. *DEV Community*. https://dev.to/litmus-chaos/github-chaos-actions-in-your-ci-cd-workflow-mke

GeeksforGeeks, & achal11sp. (2024, August 29). 10 microservices design principles that every developer should know. https://www.geeksforgeeks.org/10-microservices-design-principles-that-every-developer-should-know/

GeeksforGeeks, & gpancomputer. (2024, August 1). Fault tolerance in distributed system. https://www.geeksforgeeks.org/fault-tolerance-in-distributed-system/

GeeksforGeeks, & navlaniwesr. (2024, December 5). Monolithic architecture system design. https://www.geeksforgeeks.org/monolithic-architecture-system-design/

GeeksforGeeks, & Rawat, T. (2025, January 3). What are microservices? https://www.geeksforgeeks.org/microservices

GeeksforGeeks, & Venugopal, S. (2024, May 31). The story of Netflix and microservices. https://www.geeksforgeeks.org/the-story-of-netflix-and-microservices/

GitHub. (n.d.-a). GitHub actions. Retrieved April 6, 2025, from https://github.com/features/actions

GitHub. (n.d.-b). Deploying with GitHub actions. Retrieved April 8, 2025, from https://docs.github.com/en/actions/use-cases-and-examples/deploying/deploying-with-github-actions

GitLab. (2022, September 29). What are the benefits of a microservices architecture? https://about.gitlab.com/blog/2022/09/29/what-are-the-benefits-of-a-microservices-architecture/

Goikhman, E. (2024, April 4). Chaos testing: The ultimate guide. *Vulcan Cyber*. https://vulcan.io/blog/chaos-testing-what-you-need-to-know/

Google Cloud. (n.d.). What is Kubernetes? Retrieved April 11, 2025, from https://cloud.google.com/learn/what-is-kubernetes

Gremlin. (n.d.). Gremlin. Retrieved April 11, 2025, from https://www.gremlin.com/solutions/shift-left-reliability-testing

Gremlin. (2023a, June 28). Comparing chaos engineering tools. https://www.gremlin.com/community/tutorials/chaos-engineering-tools-comparison

Gremlin. (2023b, October 12). Chaos engineering: The history, principles, and practice. https://www.gremlin.com/community/tutorials/chaos-engineering-the-history-principles-and-practice

Gunja, S. (2024, April 29). What is chaos engineering? *Dynatrace News*. https://www.dynatrace.com/news/blog/what-is-chaos-engineering/

Harness, & Gaikwad, C. (2024, July 24). Top benefits of continuous integration. https://www.harness.io/blog/benefits-of-continuous-integration

Harness, & Minick, E. (2024, October 18). Top benefits of continuous integration. https://www.harness.io/blog/benefits-of-continuous-integration

Hicks, M. (2025, March 19). Turn chaos into confidence: 5 benefits of chaos engineering. *OpenText Blogs*. https://blogs.opentext.com/turn-chaos-into-confidence-5-benefits-of-chaos-engineering/

Hochstetler, A. H. (2024, February 8). The evolution of microservices: Trends and predictions for the future. *The Architect Guild*. https://thearchitectguild.com/2024/02/08/the-evolution-of-microservices-trends-and-predictions-for-the-future/

IBM. (2023, August 3). Chaos engineering. *IBM Think*. https://www.ibm.com/think/topics/chaos-engineering

InfoQ. (2017, February 22). Mastering chaos - A Netflix guide to microservices [Video]. YouTube. https://www.youtube.com/watch?v=CZ3wIuvmHeM

Jamesmontemagno. (2022a, April 13). Data sovereignty per microservice - .NET. *Microsoft Learn*. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/data-sovereignty-per-microservice

Jamesmontemagno. (2022b, April 13). Service-oriented architecture - .NET. *Microsoft Learn*. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/service-oriented-architecture

Joyce, J. E., & Kothamasu, D. (2024, November 15). Security governance simplified: Protecting your microservice applications. *DZone*. https://dzone.com/articles/security-governance-simplified-protecting-your-mic

Kong. (n.d.-a). Essential guide to understanding microservices architecture. Retrieved April 4, 2025, from https://konghq.com/blog/learning-center/what-are-microservices

Kong. (n.d.-b). Exploring SOA: What is service-oriented architecture? Retrieved April 4, 2025, from https://konghq.com/blog/learning-center/soa-in-web-services

Kong. (n.d.-c). Understanding service discovery for microservices architecture. Retrieved April 4, 2025, from https://konghq.com/blog/learning-center/service-discovery-in-a-microservices-architecture

Krishnan, R. (2024, November 14). Microservices — A definitive guide. *Medium*. https://solutionsarchitecture.medium.com/microservices-a-definitive-guide-7ebae643be8e

Krzywiec, W. (2020, August 28). Introduction to Kubernetes: What problems does it solve? *Medium*. https://wkrzywiec.medium.com/introduction-to-kubernetes-what-problems-does-it-solve-8a72400cfb2e

Kubecost. (n.d.). Kubernetes distributions: Tutorial & explanation. Retrieved April 11, 2025, from https://www.kubecost.com/kubernetes-multi-cloud/kubernetes-distributions/

Laoyan, S. (2025, February 20). What is agile methodology? (A beginner's guide) [2025]. *Asana*. https://asana.com/resources/agile-methodology

Linux Screenshots. (2016, January 25). Grafana dashboard [Photograph]. Flickr. https://www.flickr.com/photos/xmodulo/24311604930/

Litmus Chaos. (n.d.). Collaborate with teams. Retrieved April 6, 2025, from https://docs.litmuschaos.io/docs/concepts/teaming

Lüders, R. (2025, April 8). Run a lightweight local Kubernetes cluster with K3s. *Medium*. https://medium.com/@rluders/run-a-lightweight-local-kubernetes-cluster-with-k3s-1879f8a2b023

Lunbeck, N. (2024, July 3). K0s vs K3s vs K8s: Comparing Kubernetes distributions. *Shipyard*. https://shipyard.build/blog/k0s-k3s-k8s/

M, S. (2024, November 16). The rise of microservices: Why they are shaping the future of software development. *Medium*. https://medium.com/@miralashiva27/the-rise-of-microservices-why-they-are-shaping-the-future-of-software-development-ecdf0fef6d6b

Manasa, S. (2024, November 15). Understanding Kubernetes architecture. *Medium*. https://saimanasak.medium.com/understanding-kubernetes-architecture-add159d720fd

Mannotra, V. (2025, January 14). Understanding end-to-end microservices testing. *BrowserStack*. https://www.browserstack.com/guide/end-to-end-testing-in-microservices

MCSGA. (2024, November 29). Agile methodology: Flexibility and efficiency in project management. *Medium*. https://medium.com/@MakeComputerScienceGreatAgain/agile-methodology-flexibility-and-efficiency-in-project-management-eda8bc1c6201

Mr.PlanB. (2024, November 17). etcd 101, understanding its role in Kubernetes architecture. *Medium*. https://medium.com/@PlanB./etcd-101-understanding-its-role-in-kubernetes-architecture-bb9c597908c2

Muppeda, A. (2024, November 24). A hands-on guide to Kubernetes horizontal & vertical pod autoscalers. *Medium*. https://medium.com/@muppedaanvesh/a-hands-on-guide-to-kubernetes-horizontal-vertical-pod-autoscalers-%EF%B8%8F-58903382ef71

Nayak, S. (2025, February 4). Docker vs Kubernetes: Key differences in containerization and orchestration. *CloudOptimo*. https://www.cloudoptimo.com/blog/docker-vs-kubernetes-key-differences-in-containerization-and-orchestration/

Nazarenko, O. (2025, February 14). Top 5 SDLC models for effective project management. *MindK*. https://www.mindk.com/blog/sdlc-models/

Nazaryan, T. (2024, September 13). 6 best Linux distro for server hosting. *10Web*. https://10web.io/blog/best-linux-distro-for-server-hosting/

Ndungu, S. (2022, February 7). Jenkins automation: What it is & how it works. *BlazeMeter*. https://www.blazemeter.com/blog/jenkins-automation

NetflixOSS. (n.d.). Chaos monkey. Retrieved April 6, 2025, from https://netflix.github.io/chaosmonkey/

Novet, J. (2017, March 1). AWS is investigating S3 issues, affecting Quora, Slack, Trello (updated). *VentureBeat*. https://venturebeat.com/business/aws-is-investigating-s3-issues-affecting-quora-slack-trello/

Nuageup Team. (2024, November 17). Choosing the right Kubernetes distribution for your organization. *Nuageup*. https://nuageup.com/blog/choosing-the-right-kubernetes-distribution-for-your-organization

Nutanix. (2020, February 14). The cost of unplanned IT downtime & outages. https://www.nutanix.com/uk/blog/the-cost-of-downtime

Nvisia Learn. (2023, March 30). The agile process 101: Understanding the benefits of using agile methodology. *Nvisia*. https://www.nvisia.com/insights/agile-methodology

Okta. (n.d.). 8 ways to secure your microservices architecture. Retrieved April 5, 2025, from https://www.okta.com/resources/whitepaper/8-ways-to-secure-your-microservices-architecture/

OpenText. (n.d.). What is chaos engineering? Retrieved April 1, 2025, from https://www.opentext.com/what-is/chaos-engineering

Oracle. (n.d.). Service-oriented architecture (SOA) and web services: The road to enterprise application integration (EAI). Retrieved April 3, 2025, from https://www.oracle.com/technical-resources/articles/javase/soa.html

Oso. (n.d.). 10 microservices best practices. Retrieved April 2, 2025, from https://www.osohq.com/learn/microservices-best-practices

Palachi, E. (2025, March 7). Monolith to microservices: All you need to know. *vFunction*. https://vfunction.com/blog/monolith-to-microservices/

Palle, R. (2024, July 8). Ensuring data integrity: The role of ACID transactions in modern distributed microservices architecture. *IEEE Computer Society*. https://www.computer.org/publications/tech-news/community-voices/acid-transactions-in-distributed-microservices-architecture

Palo Alto Networks. (n.d.-a). What are microservices? Retrieved April 5, 2025, from https://www.paloaltonetworks.com/cyberpedia/what-are-microservices

Palo Alto Networks. (n.d.-b). What is the CI/CD pipeline? Retrieved April 5, 2025, from https://www.paloaltonetworks.com/cyberpedia/what-is-the-ci-cd-pipeline-and-ci-cd-security

Pandey, A. K. (2024, October 30). What is CI/CD? Continuous integration and continuous deployment. *AtulHost*. https://www.atulhost.com/what-is-ci-cd

Parashar, P. (2024, July 25). Understanding cloud outages: Causes, consequences and mitigation strategies. *HCLTech*. https://www.hcltech.com/trends-and-insights/understanding-cloud-outages-causes-consequences-and-mitigation-strategies

Patel, A. (2024, August 12). Kubernetes — Architecture and cluster components overview. *Medium*. https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding-k8s-cluster-components-90e11eb34ccd

Peck, N. (2018a, June 13). Microservice principles: Decentralized governance. *Medium*. https://medium.com/@nathankpeck/microservice-principles-decentralized-governance-4cdbde2ff6ca

Peck, N. (2018b, June 21). Microservice principles: Decentralized data management. *Medium*. https://medium.com/@nathankpeck/microservice-principles-decentralized-data-management-4adaceea173f

Peefy. (2024, January 2). 10 ways for Kubernetes declarative configuration management. *DEV Community*. https://dev.to/peefy/10-ways-for-kubernetes-declarative-configuration-management-5pb

Powell, R. (2021, October 6). SOA vs microservices: Going beyond the monolith. *CircleCI*. https://circleci.com/blog/soa-vs-microservices/

Prajapati, K. (2024, November 16). Kubernetes: Pod creation flow. *Medium*. https://kamsjec.medium.com/kubernetes-pod-creation-flow-96b5e3543c76

Radwell, O. (2021, May 19). What's the best Kubernetes distribution for local environments? *Oliver Radwell Blog*. https://blog.radwell.codes/2021/05/best-kubernetes-distribution-for-local-environments/

Rao, C. N. (2023, September 2). The rise of microservices: A brief history. *Medium*. https://medium.com/@chetan.nrao/the-rise-of-microservices-a-brief-history-7933c4673c86

Red Hat. (2020, July 27). What is service-oriented architecture? https://www.redhat.com/en/topics/cloud-native-apps/what-is-service-oriented-architecture

Red Hat. (2023a, May 1). What are microservices? https://www.redhat.com/en/topics/microservices/what-are-microservices

Red Hat. (2023b, December 12). What is CI/CD? https://www.redhat.com/en/topics/devops/what-is-ci-cd

Red Hat. (2025, March 27). What is GitOps? https://www.redhat.com/en/topics/devops/what-is-gitops

Richardson, C. (n.d.). Pattern: Database per service. *microservices.io*. Retrieved April 4, 2025, from https://microservices.io/patterns/data/database-per-service.html

Richardson, C., & F5. (2015, May 19). Introduction to microservices. *F5, Inc*. https://www.f5.com/company/blog/nginx/introduction-to-microservices

Ricky. (2023, January 16). Kubernetes 101: An introduction to container orchestration. *FISClouds*. https://www.fisclouds.com/kubernetes-101-an-introduction-to-container-orchestration-9265/

Rob Bagby. (n.d.). Backends for frontends pattern - Azure architecture center. *Microsoft Learn*. Retrieved April 4, 2025, from https://learn.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends

Sahoo, J. (2025, March 19). Optimizing Kubernetes deployments: CI/CD pipeline essentials. *DevTron*. https://devtron.ai/blog/ci-cd-pipeline-for-kubernetes/

Schillerstrom, M. (2022, October 11). The top chaos engineering tools. *Harness*. https://www.harness.io/blog/chaos-engineering-tools

Singh, R. (2025, March 28). Waterfall methodology. *Institute of Project Management*. https://projectmanagement.ie/blog/waterfall-methodology

Solo.io. (n.d.). What are microservices? Architecture, challenges, and tips. Retrieved April 3, 2025, from https://www.solo.io/topics/microservices

Sonar, V. (2024, November 27). How to integrate chaos engineering into CI/CD. *Aviator Blog*. https://www.aviator.co/blog/how-to-integrate-chaos-engineering-into-your-ci-cd-pipeline

Sonaniya, D. (2025, March 11). Chaos engineering: Preparing for the unpredictable in CI/CD. *Encanto Technologies*. https://encantotek.com/chaos-engineering-preparing-for-the-unpredictable-in-ci-cd/

Soni, R. (2025, February 2). Monitoring and observability in microservices: A comprehensive guide with practical examples. *Medium*. https://medium.com/javarevisited/monitoring-and-observability-in-microservices-a-comprehensive-guide-with-practical-examples-7b43937d890d

Sparkfabrik. (2021, November 11). GitOps and Kubernetes: CI/CD for cloud native applications. https://blog.sparkfabrik.com/en/gitops-and-kubernetes

Spot.Io. (2024, November 15). Kubernetes architecture: Control plane, data plane, components. https://spot.io/resources/kubernetes-architecture/11-core-components-explained/

Steadybit. (2024a, August 6). Enhance access control in chaos engineering with steadybit. https://steadybit.com/blog/blast-radius-and-access-control-strategies-for-a-safer-system/

Steadybit. (2024b, September 23). Chaos engineering: Strengthen systems with controlled failures. https://steadybit.com/blog/chaos-engineering-a-beginners-guide/

Swimm Team. (2024, May 13). Microservices monitoring: Importance, metrics & 5 best practices. *Swimm*. https://swimm.io/learn/microservices/microservices-monitoring-importance-metrics-and-5-critical-best-practices

Tanner, M. (2024, September 14). What is a monolithic application? Everything you need to know. *vFunction*. https://vfunction.com/blog/what-is-monolithic-application/

Tetteh, I. (2025, March 17). Designing reliable CI CD pipelines for faster, error-free software releases. *Ambassador*. https://www.getambassador.io/blog/reliable-ci-cd-pipelines-faster-software-releases

Tieturi. (2025, February 7). The benefits of Kubernetes as a platform for modern software development. https://www.tieturi.fi/en/blogi/the-benefits-of-kubernetes-as-a-platform-for-modern-software-development/

Tripathy, A. (2021, December 7). Understanding Kubernetes cluster autoscaling. *Medium*. https://medium.com/kubecost/understanding-kubernetes-cluster-autoscaling-675099a1db92

Udasi, A. (2024, September 19). How chaos engineering boosts system stability. *Zenduty*. https://zenduty.com/blog/chaos-engineering/

Virtuoso QA. (n.d.). The 7 principles of microservices. Retrieved April 3, 2025, from https://www.virtuosoqa.com/post/the-7-principles-of-microservices

Wickramasinghe, S. (2023, July 17). Chaos engineering: Benefits, best practices, and challenges. *Splunk*. https://www.splunk.com/en_us/blog/learn/chaos-engineering.html

Wickramasinghe, S. (2024, July 19). Chaos testing explained. *Splunk*. https://www.splunk.com/en_us/blog/learn/chaos-testing.html

Wikipedia contributors. (2024, October 14). V-model (software development). *Wikipedia*. https://en.wikipedia.org/wiki/V-model_(software_development)

Wikipedia contributors. (2025, January 22). Polyglot persistence. *Wikipedia*. https://en.wikipedia.org/wiki/Polyglot_persistence

Appendix 1: Data management plan

Description of the thesis material

The research material for this thesis consists of data collected through means of an online questionnaire. The questionnaire was designed by the author of the research work to gather information relevant to the thesis topic.

The primary method of material collection was survey research, specifically the use of an online questionnaire distributed to the target audience. The target audience for the questionnaire consisted of two categories: students and IT professionals currently working in the industry. No personally identifiable information (PII) such as names or email addresses, to ensure the anonymity of the participants. Only general background information such as role (student or professional), years of experience, highest degree achieved, or area of studies was collected to help the responses in proper context. No previously collected third-party material was used as the primary source of information for this thesis. The questionnaire and the gathered responses are original to this research. The data was digitally collected, processed and stored in both formats, XSLX and CSV.

Management and storage of the research data

The research material is stored and processed on the thesis author's own computer, protected by multi-factor authentication (MFA). The raw anonymous data exported from the responses is stored in a dedicated folder for the research purposes. The data is stored redundantly on two different storage drives.

Processing of personal data and sensitive data

This thesis does not process personally identifiable information or sensitive data. The questionnaire used was designed to guarantee anonymity to all respondents. While some background information was collected, this information was collected in a way that it does not allow the identification of individual respondents. The main purpose of collecting the background information was to have a general description of the respondent pool of students and IT professionals. The participants were informed at the beginning of the questionnaire and before answering any questions about the anonymity of the survey.

Material ownership

The ownership of the research material collected for the thesis resides with the author of the thesis. There is no external party or client commissioning the material, which means ownership rights are not shared. Copyright of the written document, including analysis and interpretation of the resulting data, also belongs to the author of this work.

Further use of the material after the work is completed

The research work, data and material collected for this thesis will not be made available for further use. The author of the thesis will keep the anonymised data securely stored for one year from the date of acceptance of the thesis. This is to ensure that the results of the thesis and survey work conducted can be verified should that be necessary. After this one-year period, the material will be disposed of in a secure manner by means of permanent deletion.

Appendix 2. Vagrantfile Virtual Machine Script

```
Vagrant.configure("2") do |config|
  config.vm.box = "generic/ubuntu2204"
  config.vm.network "private_network", ip: "192.168.33.10"
  config.vm.synced_folder "./k3s", "/k3s"
  config.vm.disk :disk, primary: true, size: "10GB"

  config.vm.provider "virtualbox" do |vb|
    # Customize the amount of cpu cores on the VM:
    vb.cpus = "4"

    # Customize the amount of memory on the VM:
    vb.memory = "8192"
  end
end
```

Appendix 3. Example Kubernetes Deployment File

```yaml
apiVersion: v1
kind: Service
metadata:
    name: user-service
    namespace: wallet
    labels:
        app: user-service
spec:
    ports:
        - port: 8081
          targetPort: 8081
          name: http
    selector:
        app: user-service
---
apiVersion: apps/v1
kind: Deployment
metadata:
    name: user-service
    namespace: wallet
spec:
    selector:
        matchLabels:
            app: user-service
    replicas: 1
    template:
        metadata:
            labels:
                app: user-service
        spec:
            containers:
                - name: user-service
                  image: ahm282/user-service:latest
                  imagePullPolicy: IfNotPresent
                  ports:
                      - containerPort: 8081
                  env:
                      - name: USER_DB_HOST
                        valueFrom:
                            configMapKeyRef:
                                name: wallet-config
                                key: USER_DB_HOST
```

```
            - name: USER_DB_PORT
              valueFrom:
                  configMapKeyRef:
                      name: wallet-config
                      key: USER_DB_PORT
            - name: USER_DB_NAME
              valueFrom:
                  configMapKeyRef:
                      name: wallet-config
                      key: USER_DB_NAME
            - name: USER_DB_USER
              valueFrom:
                  secretKeyRef:
                      name: wallet-secrets
                      key: USER_DB_USER
            - name: USER_DB_PASS
              valueFrom:
                  secretKeyRef:
                      name: wallet-secrets
                      key: USER_DB_PASS
        resources:
            requests:
                memory: "256Mi"
                cpu: "100m"
            limits:
                memory: "512Mi"
                cpu: "200m"
```

Appendix 4. Setup Automation Bash Script

```
#!/bin/bash
set -e

# Function to print ASCII art header
print_header() {
cat << "EOF"


 _  ___________                ____ _
| |/ /___ / ___|              / ___| |__   __ _  ___  ___
| ' /  |_ ¥___ ¥   _____    | |   | '_ ¥ / _` |/ _ ¥/ __|
| . ¥ ___) |__) | |_____| | |___| | | | (_| | (_) ¥__ ¥
|_|¥_¥____/____/           ¥____|_| |_|¥__,_|¥___/|___/
 _____                _                        _
| ____|_ __   __ _(_)_ __   ___  ___ _ __(_)_ __   __ _
|  _| | '_ ¥ / _` | | '_ ¥ / _ ¥/ _ ¥ '__| | '_ ¥ / _` |
| |___| | | | (_| | | | | |  __/  __/ |  | | | | | (_| |
|_____|_| |_|¥__, |_|_| |_|¥___|¥___|_|  |_|_| |_|¥__, |
             |___/                                  |___/

EOF
}

# Function to check if a command exists
command_exists() {
    command -v "$1" >/dev/null 2>&1
}

# Check for sudo permissions without password prompt
if ! sudo -n true 2>/dev/null; then
    echo "⚠  Please ensure you have sudo privileges (and that sudo does not require
a password for this session)."
    exit 1
fi

# Display the header
print_header

################################
# Install K3s
################################
read -p "Would you like to install K3s? (Y/n): " choice
choice=${choice:-y}
```

```
if [[ "$choice" =~ ^[Yy]$ ]]; then
    echo "🚀 Installing K3s..."
    curl -sfL https://get.k3s.io | sh -s - --node-ip=192.168.33.10 --advertise-
address=192.168.33.10 --flannel-backend=vxlan
    echo "✅ K3s installed."

    echo "🔗 Waiting for Kubernetes API to stabilize..."
    sleep 5

    echo "Copying Kubeconfig to ~/.kube/config..."
    if [ ! -d ~/.kube ]; then
        mkdir -p ~/.kube
    fi

    if [ -f /etc/rancher/k3s/k3s.yaml ]; then
        sudo cp /etc/rancher/k3s/k3s.yaml ~/.kube/config
        echo "✅ Kubeconfig copied to ~/.kube/config."

        echo "🔧 Setting permissions for Kubeconfig..."
        sudo chown $(id -u):$(id -g) ~/.kube/config
        sudo chmod 644 ~/.kube/config

        echo "🔧 Setting KUBECONFIG environment variable..."
        export KUBECONFIG=~/.kube/config
        echo 'export KUBECONFIG=~/.kube/config' >> ~/.bashrc
        source ~/.bashrc

        echo "🎉 K3s setup complete."
    else
        echo "❌ Kubeconfig not found at /etc/rancher/k3s/k3s.yaml. Please check
your K3s installation."
        exit 1
    fi
fi

##############################
# Run deploy script
##############################
read -p "Would you like to run the deploy script now? (Y/n): " choice
choice=${choice:-y}
if [[ "$choice" =~ ^[Yy]$ ]]; then
    if [ -x "./k3s-deploy.sh" ]; then
        echo "Running deploy script..."
        ./k3s-deploy.sh
```

```
    else
        echo "❌ Deploy script not found or not executable. Ensure './k3s-deploy.sh'
exists and has execution permissions."
    fi
else
    echo "You can run the deploy script later by executing: ./deploy.sh"
    echo ""
fi

##############################
# Install Chaos Mesh
##############################
read -p "Would you like to install Chaos Mesh for chaos engineering? (Y/n): " choice
choice=${choice:-y}
if [[ "$choice" =~ ^[Yy]$ ]]; then
    echo "📦 Installing Helm (if not already installed)..."
    if ! command_exists helm; then
        curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 |
bash
    else
        echo "✅ Helm is already installed."
    fi

    echo "📦 Installing Chaos Mesh..."
    helm repo add chaos-mesh https://charts.chaos-mesh.org
    helm repo update

    kubectl create namespace chaos-mesh || echo "Namespace chaos-mesh already
exists."
    helm install chaos-mesh chaos-mesh/chaos-mesh -n chaos-mesh ¥
        --set chaosDaemon.runtime=containerd ¥
        --set chaosDaemon.socketPath=/run/k3s/containerd/containerd.sock ¥
        --version 2.7.1 ¥
        --set metrics.enable=true ¥
        --set metrics.serviceMonitor.enable=true ¥
        --set dashboard.securityMode=false

    # Create a ClusterRoleBinding for Chaos Mesh !!!GRANTS FULL ACCESS RBAC!!!
cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: chaos-mesh-full-access
subjects:
```

```
  - kind: ServiceAccount
    name: chaos-mesh-controller-manager
    namespace: chaos-mesh
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
EOF


    kubectl get po -n chaos-mesh
    kubectl get svc -n chaos-mesh chaos-dashboard

    echo "🎉 Chaos Mesh installed successfully."
    DASHBOARD_PORT=$(kubectl get svc -n chaos-mesh chaos-dashboard -o
jsonpath='{.spec.ports[0].nodePort}')
    echo "🔑 Access the dashboard using: http://localhost:${DASHBOARD_PORT}"
fi

##############################
# Install Prometheus and Grafana
##############################
read -p "Would you like to install Prometheus and Grafana for monitoring? (Y/n): "
choice
choice=${choice:-y}
if [[ "$choice" =~ ^[Yy]$ ]]; then
    echo "📊 Installing Prometheus and Grafana..."

    echo "🔍 Checking for Helm..."
    if ! command_exists helm; then
        echo "Helm is not installed. Installing Helm..."
        curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 |
bash
    else
        echo "✅ Helm is already installed."
    fi

    echo "📦 Adding Prometheus and Grafana Helm repositories..."
    helm repo add prometheus-community https://prometheus-community.github.io/helm-
charts
    helm repo add grafana https://grafana.github.io/helm-charts
    helm repo update

    echo "📦 Creating monitoring namespace..."
```

```
    kubectl create namespace monitoring || echo "Namespace monitoring already
exists."

    echo "🔍 Installing Prometheus..."
    helm install prometheus prometheus-community/kube-prometheus-stack ¥
        --namespace monitoring ¥
        --set alertmanager.enabled=false ¥
        --set pushgateway.enabled=false

    echo "🔑 Installing Grafana..."
    echo " Creating Grafana dashboards configmap..."
    kubectl create configmap grafana-dashboards -n monitoring --from-
file=./grafana/dashboards

    helm install grafana grafana/grafana -n monitoring -f ./grafana/values.yaml

    echo "🎉 Prometheus and Grafana installed successfully."

    # Get the node eth1 IPv4 address and Grafana NodePort
    GRAFANA_PORT=$(kubectl get svc grafana -n monitoring -o
jsonpath='{.spec.ports[0].nodePort}')
    NODE_IP=$(ip a show eth1 | grep -oP '(?<=inet¥s)¥d+(¥.¥d+){3}')
    echo "🔑 Access Grafana dashboard at: http://${NODE_IP}:${GRAFANA_PORT}"
    echo "📝 Default login credentials:"
    echo "   Username: admin"
    echo "   Password: admin (change upon first login)"
    kubectl get pods -n monitoring
fi

##############################
# Install Loki and Promtail
##############################
read -p "Would you like to install Loki for log aggregation? (Y/n): " choice
choice=${choice:-y}
if [[ "$choice" =~ ^[Yy]$ ]]; then
    echo "📦 Installing Helm (if not already installed)..."
    if ! command_exists helm; then
        curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 |
bash
    else
        echo "✅ Helm is already installed."
    fi

    echo "📦 Adding Grafana Helm repository (if not already added)..."
```

```
    helm repo add grafana https://grafana.github.io/helm-charts
    helm repo update

    echo "📦 Creating Monitoring namespace..."
    kubectl create namespace monitoring || echo "Namespace 'monitoring' already
exists."

    echo "📦 Installing Loki stack (Loki + Promtail)..."
    helm install loki grafana/loki-stack ¥
        --namespace monitoring ¥
        --set loki.image.tag=2.9.13 ¥
        --set loki.persistence.enabled=true ¥
        --set loki.persistence.size=1Gi ¥
        --set loki.auth.enabled=false

    echo "🎉 Loki stack installed successfully."
    echo "To view logs or configure further, check the resources in the 'monitoring'
namespace."
fi

##############################
# Install K9s
##############################
echo "🔍 Checking for K9s..."
if ! command -v k9s &> /dev/null; then
    read -r -p "K9s is not installed. Would you like to install K9s? (Y/n): " choice
    choice=${choice:-y}

    if [[ "$choice" =~ ^[Yy]$ ]]; then
        echo "📦 Installing K9s dashboard..."

        # Download if not already present
        if [[ ! -f k9s_linux_amd64.deb ]]; then
            wget
https://github.com/derailed/k9s/releases/latest/download/k9s_linux_amd64.deb
        else
            echo "✅ K9s binary already exists. Skipping download."
        fi

        # Install and clean up
        sudo dpkg -i k9s_linux_amd64.deb
        rm -f k9s_linux_amd64.deb
    fi
else
```

```
    echo "✅ K9s is already installed."
fi

##############################
# Run Ngrok
##############################
echo "🔍 Checking for Ngrok..."
if ! command_exists ngrok; then
    read -p "Ngrok is not installed. Would you like to install Ngrok (Y/n): " choice
    choice=${choice:-y}
    if [[ "$choice" =~ ^[Yy]$ ]]; then
        echo "📦 Installing Ngrok..."
        curl -sSL https://ngrok-agent.s3.amazonaws.com/ngrok.asc ¥
        | sudo tee /etc/apt/trusted.gpg.d/ngrok.asc >/dev/null ¥
        && echo "deb https://ngrok-agent.s3.amazonaws.com buster main" ¥
        | sudo tee /etc/apt/sources.list.d/ngrok.list ¥
        && sudo apt update ¥
        && sudo apt install ngrok

        echo "✅ Ngrok installed. Please run 'ngrok authtoken <your-auth-token>' to
authenticate before using Ngrok."
    fi
else
    echo "✅ Ngrok is already installed."
    read -p "Would you like to run Ngrok to expose the application? (Y/n): " choice
    choice=${choice:-y}
    if [[ "$choice" =~ ^[Yy]$ ]]; then
        echo "Running Ngrok..."
        ngrok http --url=wasp-concrete-excited.ngrok-free.app 127.0.0.1:80
    fi
fi

echo "🎉 All selected components have been installed and configured."
echo "🚀 Enjoy your K3s setup!"
```

Appendix 5. Stress Reliability Tests

```yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: Workflow
metadata:
    name: stress-reliability-test-v3
    namespace: wallet
spec:
    entry: service-reliability-tests
    templates:
        - name: service-reliability-tests
          templateType: Serial
          children:
              - pod-failure-experiments
              - network-disruption-experiments
              - resource-constraint-experiments
          deadline: 15m
        - name: pod-failure-experiments
          templateType: Serial
          children:
              - user-service-pod-failure
              - finance-service-pod-failure
        - name: user-service-pod-failure
          templateType: PodChaos
          deadline: 1m
          podChaos:
              action: pod-failure
              selector:
                  namespaces:
                      - wallet
                  labelSelectors:
                      app: user-service
              mode: one
        - name: finance-service-pod-failure
          templateType: PodChaos
          deadline: 1m
          podChaos:
              action: pod-failure
              selector:
                  namespaces:
                      - wallet
                  labelSelectors:
                      app: finance-service
              mode: one
```

```
    - name: network-disruption-experiments
      templateType: Serial
      children:
        - api-gateway-network-chaos
        - inter-service-network-partition
    - name: api-gateway-network-chaos
      templateType: NetworkChaos
      deadline: 3m
      networkChaos:
        action: delay
        target:
          mode: all
        selector:
          namespaces:
            - wallet
          labelSelectors:
            app: api-gateway
        mode: all
        delay:
          latency: 500ms
          correlation: "80"
          jitter: 100ms
    - name: inter-service-network-partition
      templateType: NetworkChaos
      deadline: 3m
      networkChaos:
        action: partition
        target:
          mode: all
        selector:
          namespaces:
            - wallet
        mode: all
    - name: resource-constraint-experiments
      templateType: Serial
      children:
        - user-db-resource-stress
        - finance-db-resource-stress
    - name: user-db-resource-stress
      templateType: StressChaos
      deadline: 3m
      stressChaos:
        selector:
          namespaces:
```

```
                - wallet
            labelSelectors:
                app: user-db
        mode: one
        stressors:
            memory:
                workers: 2
                size: 1G
    - name: finance-db-resource-stress
      templateType: StressChaos
      deadline: 3m
      stressChaos:
        selector:
            namespaces:
                - wallet
            labelSelectors:
                app: finance-db
        mode: one
        stressors:
            cpu:
                workers: 2
                load: 80
```

Appendix 6. Chaos Engineering Questionnaire Responses

This appendix presents the aggregated results from the survey conducted for this thesis. A total of N=25 individuals participated in the survey.

6.1 Demographics of Survey Respondents

6.1.1 Age Distribution

- 18-20 years: 13 respondents (52%)
- 21-23 years: 6 respondents (24%)
- 24-26 years: 1 respondent (4%)
- 27-29 years: 1 respondent (4%)
- 30+ years: 4 respondents (16%)

6.1.2 Education Level

- 1st year - Bachelor's: 4 respondents (16%)
- 2nd year - Bachelor's: 8 respondents (32%)
- 3rd year - Bachelor's: 9 respondents (36%)
- Associate degree: 1 respondent (4%)
- High School: 1 respondent (4%)
- No answer: 2 respondents (8%)

6.1.3 Work Experience

- No prior work experience: 13 respondents (52%)
- Prior work experience: 12 respondents (48%)
    - *Distribution among those with experience:*
        - Less than 1 year: 4 respondents
        - 1-2 years: 3 respondents
        - 3-5 years: 1 respondent
        - 6+ years: 3 respondents
        - 3+ years: 1 respondent *(Note: This category might overlap with previous ones)*

6.1.4 Current Role

- Student: 22 respondents (88%)
- Teacher: 1 respondent (4%)
- Industry Professional: 2 respondents (8%)

6.2 IT Interests and Technical Familiarity

6.2.1 IT Interests (Students, N=22)

- Application Development: 7 respondents
- Cybersecurity: 5 respondents
- Cloud and DevOps: 3 respondents
- Artificial Intelligence (AI): 2 respondents
- Other/Multiple interests: 5 respondents

6.2.2 Familiarity with Chaos Engineering

- Not at all familiar: 15 respondents (60%)
- Basic familiarity: 3 respondents (12%)
- Intermediate familiarity: 3 respondents (12%)
- No answer: 4 respondents (16%)

6.2.3 Familiarity with Microservices

- No: 8 respondents (32%)
- Yes, to some extent: 11 respondents (44%)
- Yes, extensively: 2 respondents (8%)
- No answer: 4 respondents (16%)

6.2.4 Familiarity with CI/CD (Continuous Integration/Continuous Deployment)

- No: 6 respondents (24%)
- Yes, to some extent: 13 respondents (52%)
- Yes, extensively: 3 respondents (12%)
- No answer: 3 respondents (12%)

6.2.5 Prior Education on Chaos Engineering

- No: 17 respondents (68%)
- Yes, to some extent: 5 respondents (20%)
- No answer: 3 respondents (12%)

6.3 Technology Usage Experience

6.3.1 Experience with Software Deployment

- No: 14 respondents (56%)
- Yes: 7 respondents (28%)
- No answer: 4 respondents (16%)

6.3.2 Experience Using Microservices

- No: 14 respondents (56%)
- Yes: 4 respondents (16%)
- No answer: 7 respondents (28%)

6.3.3 Experience Using CI/CD

- No: 14 respondents (56%)
- Yes: 7 respondents (28%)
- No answer: 4 respondents (16%)

6.3.4 Experience Using Chaos Engineering

- No: 18 respondents (72%)
- Yes: 1 respondent (4%)
- No answer: 6 respondents (24%)

6.4 Interest and Perceptions Regarding Chaos Engineering

6.4.1 Interest in Learning More About Chaos Engineering

- No: 5 respondents (20%)
- Maybe: 9 respondents (36%)
- Yes: 8 respondents (32%)
- No answer: 3 respondents (12%)

6.4.2 Perceived Importance of Chaos Engineering (Industry Professionals and Teachers, N=3)

- Neutral: 1 respondent
- Extremely impactful: 2 respondents
- *(Students (N=22) were not asked this question)*

6.4.3 Perceived Risks of Not Implementing Chaos Engineering (Multiple Selections Allowed)

- Reduced system resilience: 11 respondents
- Increased downtime and outages: 9 respondents
- I am not familiar with chaos engineering: 8 respondents
- Not sure / no opinion: 6 respondents
- Poor customer experience: 4 respondents
- Loss of competitive advantage: 3 respondents
- Increased operational costs: 2 respondents

6.4.4 Perceived Future of Chaos Engineering

- Not sure / no opinion: 12 respondents
- It will gain popularity but face resistance: 6 respondents
- It will become a standard practice: 3 respondents
- It will remain a niche practice: 3 respondents
- There's no need for chaos engineering: 1 respondent

6.4.5 Desired Support Resources for Adopting Chaos Engineering (Multiple Selections Allowed)

- Case Studies: 11 respondents
- Training Programs: 10 respondents
- Tooling Recommendations: 9 respondents
- Not sure / no opinion: 8 respondents
- Technical Documentation: 7 respondents
- Cost-Benefit Analysis: 2 respondents