Herausragende Masterarbeiten

Autor*in

Engin Yöyen

Studiengang

Software Engineering for Embedded Systems, M.Eng.

Masterarbeitstitel

Architecture of safety-critical applications running in the public cloud

Distance and Independent Studies Center
DISC

Rheinland-Pfälzische Technische Universität

Kaiserslautern-Landau

Distance Study Program

Software Engineering for Embedded Systems

Master's Thesis

Architecture of safety-critical applications running in the public cloud

---

Provided by

Engin Yöyen

First supervisor: Prof. Dr.-Ing. Peter Liggesmeyer

Second supervisor: Dr. Rasmus Adler

Declaration

Ich versichere, dass ich diese Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Place, date	Signature
---	Engin Yöyen ---

Abstract

This master thesis presents a collection of architectural design patterns for safety-critical systems deployed on public cloud infrastructure. The research aims to enhance system reliability, mitigate risks, and improve overall performance in safety-critical applications. The study follows a systematic approach, considering multiple safety-critical use cases and prioritizing factors such as timing constraints and system resilience. The railway signaling system, particularly the moving block computation, is selected as the most suitable use case due to its ability to tolerate response delays and re-request computations. The thesis addresses four research questions concerning the deployment of safety-critical systems to the public cloud, existing fault-tolerance methods in the cloud, identification of relevant design patterns, and the applicability of design patterns in various safety-critical systems.

The study identifies and review's fault tolerance methods and cloud failure modes, which serve as a basis for identifying design patterns. The Structured What-If Technique (SWIFT) is utilized to analyze prospective hazards and recommend actions, which are then mapped onto design patterns for wide applicability across different projects. Each design pattern presents a problem statement, guidelines for implementation, and associated benefits and drawbacks.

The contribution of this thesis lies in the development of a valuable resource for architects and engineers working on safety-critical systems in the cloud. The design patterns offer practical solutions and a framework for the design and implementation of robust and secure systems. Detailed documentation, including context, benefits, drawbacks, and practical examples, facilitates understanding and adoption.

In conclusion, this thesis contributes to the advancement of safety and reliability in cloud-based safety-critical systems by providing architectural design patterns. Future research should focus on integrating security aspects, gathering diverse use cases, and validating the patterns in practical settings. Continued exploration and refinement of the design patterns will lead to more robust solutions for meeting the needs and challenges of safety-critical applications in various contexts.

Table of Contents

Abbreviations

ABS	Anti-lock Braking Systems
API	Application Programming Interface
AZ	Availability Zone
AWS	Amazon Web Services
CAP	Consistency, Availability, and Partition Tolerance
CI/CD	Continuous Integration and Continuous Delivery
CPU	Central Processing Unit
DNS	Domain Name System
ECU	Electronic Control Units
ESC	Stability Control
ERTMS	European Rail Traffic Management System
FMEA	Failure mode and effects analysis
FTA	Fault Tree Analysis
GCP	Google Cloud Platform
GSM-R	Global System for Mobile Communications - Railway
HAZOP	Hazard and Operability Study
HTTP(S)	Hypertext Transfer Protocol (Secure)
MA	Movement Authority
RBC	Radio Block Centre
RTT	Round Trip Time
SIM	Subscriber Identity Module
SLA	Service-Level-Agreement
SWIFT	The Structured What-if Technique
TCP	Transmission Control Protocol
VM	Virtual Machines

List of Tables

List of Figures

1 Introduction

Core elements that tie together our daily life and the world around us, from medical equipment to nuclear power plants, are considered safety-critical systems. The failure of these crucial elements to perform as expected carries the potential for severe consequences, endangering the well-being of individuals, the environment, or property. Therefore, such systems are considered as inherently hazardous (Knight, 2002).

Traditionally, but not exclusively, software that manages a safety-critical system is part of the system, which is deployed and run in close proximity to it, which is also called embedded systems, because it is an electronic system consisting of a hardware platform and software that is integrated in a special, well-defined technical environment and that is optimized for its specific purpose. (Marwedel, 2021). These embedded systems are designed to perform specific functions in a reliable manner with high efficiency, often controlling physical operations of the system they are embedded within. Because embedded systems actuate in the physical environment, they most often have a real-time computing constraint. For instance, airbags in most modern vehicles are programmed to inflate and deflate during a collision, in such a system, delay of the operation can cause additional injuries. Embedded systems are also limited with their processing power, memory, and connectivity.

On the opposite end of the spectrum to embedded systems, public cloud infrastructure has been trending since the launch of Elastic Compute Cloud (EC2) and Simple Storage Service (S3) by Amazon Web Services in 2006 (Surbiryala & Rong, 2019). Public cloud infrastructure is a service in which computing resources are delivered on demand by a third-party provider (e.g., Amazon Web Services, Google Cloud Platform, Microsoft Azure). On-demand access to computational resources makes it invaluable in diverse software development use cases.

The main objective of this research is to identify and evaluate different architectural design patterns that can be used to effectively address the technical challenges that arise during the development of safety-critical systems that utilize public cloud technologies, and to bridge the gap between safety-critical systems and public cloud by providing a comprehensive set of

design patterns, along with examples of how these patterns can be applied to a selected safety-critical use-case.

In this chapter, the study will be presented in a structured manner starting with a discussion on the background and context, followed by the research problem, research aims and objectives, research questions and the significance of the study. Finally, the chapter concludes by outlining the limitations of the research.

1.1 Background

A modern vehicle has around 100 Electronic Control Units (ECU), that controls everything from engine performance to airbag deployment. While some of the functionalities are there to increase the comfort of the passengers or entertain them (e.g., media player) others are safety-critical such as Anti-lock Braking Systems (ABS), Electronic Stability Control (ESC), and collision avoidance systems(Abelein et al., 2012). Heart monitors, insulin pumps, pacemakers, flight control and navigation systems, fire protection, gas detection systems are all examples of safety-critical systems that are designed to perform functions that have a direct impact on the safety and well-being of people and the environment. The potential impact of failure in safety-critical systems can be catastrophic. While a failure in a medical device such as a pacemaker can result in serious harm or even death to the patient, failure in a nuclear power plant system can result in a radioactive leak, which can cause environmental damage and pose a risk to the health of people living in the surrounding area.

Most of these safety-critical systems are considered complex systems, which consist of multiple interconnected and independent parts that interact with each other in non-linear and unpredictable ways, often exhibiting emergent behavior that cannot be easily predicted or explained by examining the individual parts alone (Bar-Yam, 2002). Interdependencies between components and their interactions make these systems difficult to design and test. In other words, the more complex a system is, the higher the likelihood of experiencing failures or unexpected behavior. Cook argues that all complex systems are inherently and unavoidably hazardous; however, it is possible to change the frequency of hazard exposure during different stages of design, development, or maintenance. The fact that these systems are hazardous drives the creation of defenses against hazards, which characterize these systems. (Cook, 1998) .

Safety-critical systems are designed with multiple layers of protection and redundancy to ensure safe and effective operation. The goal is to reduce the risk of failure or malfunction by implementing necessary measures. However, traditional approaches can be inflexible and costly, especially when considering the limited resources of embedded systems, including CPU, memory, and storage, which make it challenging to add new capabilities.

Unlike embedded systems with limited resources, in public cloud infrastructure provides access to on-demand computation power and storage. This on demand access to computational resources can be leveraged to extend the capabilities of existing safety-critical systems and discovering new methods to lower the hazardous risks associated with such systems. For instance, cloud computing can facilitate communication between vehicles (Vehicle-to-Vehicle) or infrastructure (Vehicle-to-Infrastructure) to improve safety by providing real-time information. Vehicles equipped with built-in safety features such as anti-lock braking systems (ABS) and airbags can receive additional input from the environment to enhance passenger and pedestrian safety (Wang et al., 2011).

Public cloud infrastructure refers to a service where computation resources are delivered on demand by a third-party provider (e.g., Amazon Web Services, Google Cloud Platform, Microsoft Azure). On-demand access to computation power and storage makes it invaluable in diverse business use cases. In 2021, cloud computing was embraced by a significant 41% of European union enterprises, with a primary focus on utilizing it for storing files and hosting emails. Of those enterprises, 73% utilized advanced cloud services for tasks such as hosting databases, running security software applications, and deploying computing platforms for application development and testing (*Eurostat-Cloud Computing*, 2023). Some of the key factors of the success of public cloud infrastructure are scalability, cost-efficiency, flexibility, and global reach. However, in the context of safety-critical systems running on public cloud infrastructure, issues such as communication latency, non-deterministic resource allocation, lack of standards and design becomes challenges that must be dealt with.

Hence, the main focus of this thesis is to compile a set of design patterns that can be leveraged by safety-critical systems relying on public cloud technologies. The main argument of the thesis is not that embedded systems should be rendered irrelevant, but rather that by leveraging the public cloud infrastructure, it is possible to extend the capabilities of existing safety-critical

systems, discover new methods to lower the hazardous risks associated with such systems, and achieve this in a cost-efficient manner.

1.2 Problem Statement

In the context of safety-critical systems, dependability is a fundamental property that refers to a system's capability to provide its intended functionality in a reliable and consistent manner, with the aim of ensuring that it does not pose any unacceptable risks to people, the environment, or property. Therefore, dependability encompasses quality attributes such as availability, reliability, safety, integrity, confidentiality, and maintainability (Avizienis et al., 2004). All of these quality attributes are intended to ensure that the system operates correctly under both normal and abnormal conditions. Although they are primarily mentioned in the context of safety-critical systems, they are also widely adopted in cloud computing. Therefore, there has been significant research and documentation on how to leverage these quality attributes effectively.

In the context of cloud computing, each provider extensively documents various aspects of these quality attributes, including how to increase service availability and follow best practices for data security (*AWS Well-Architected Framework*, 2023; *Google Cloud Architecture Framework*, 2023; *Microsoft Azure Well-Architected Framework*, 2023). Nevertheless, public cloud infrastructure runs mostly on commodity hardware, so failure is a common characteristic of the cloud. Although all major cloud providers offer high Service-Level-Agreement (SLA), they also clearly communicate that failure is common in cloud infrastructure.

Given the possibilities that cloud infrastructure offers, there has been a lot of interest in running safety-critical systems on the public cloud. However, there are two issues. First, there is not enough research in the area, as most of the literature focuses on possibilities rather than tangible theories that can be applied. Second, existing literature at this stage focuses on partial solutions and is not mature enough to provide extensive guidance on how to integrate safety-critical embedded systems into existing public cloud infrastructure. According to a systematic review conducted by Danielsson, Tsog, and Kunnappilly in 2018, less than 5% of the research papers reviewed between 2009 and 2016 were on real-time cloud architecture. The rest of the papers focused on topics such as scheduling, response time analysis, tools, etc. (Danielsson et al., 2018).

Based on the aforementioned issues, the following problems (P) have been identified:

- **P1 –** The area of safety-critical systems running on public cloud is relatively new and currently lacks extensive research. Further exploration in this field can lead to more practical applications and better understanding of the challenges and opportunities presented by cloud infrastructure.
- **P2** – The current research primarily concentrates on scheduling algorithms to manage timing constraints, rather than utilizing the available cloud infrastructure.
- **P3** – While cloud providers offer enough information to build resilient software, the resources in the cloud are prone to failures. However, most of the information provided is targeted towards non-safety-critical applications, leaving a gap in guidance for building resilient systems in this domain.

The limited number of practical applications for integrating safety-critical systems with cloud infrastructure can potentially slow down the growth of this area. This is because stakeholders may be discouraged from investing in the development and deployment of safety-critical systems on the public cloud without seeing enough real-world examples. Additionally, without enough scientific work, it is challenging to assess the reliability, safety, and security of safety-critical systems integrated with cloud infrastructure. This can further undermine stakeholder confidence in the technology.

Therefore, this thesis will focus on producing a set of design patterns that address failures in the cloud infrastructure (e.g., hardware, human error, network) and communication with the cloud (e.g., DNS failures), appropriate data handling, as well as software design to handle those failures.

1.3 Research Approach

In this section, the research approach will be outlined, which includes the research aim, research objectives, research questions. The initial stage of the research involves identifying the research objectives and questions. These objectives and questions have been carefully crafted to address the problems highlighted in the previous section and to guide the exploration of potential solutions.

1.3.1 Research Aim and Objectives

This thesis aims to create a collection of architectural design patterns that can be used by various safety-critical systems that run on public cloud infrastructure.

In order to achieve this aim, following research objectives are identified:

1. Identify a safety-critical use case and possible failures that might occur while running on public cloud infrastructure.
2. Investigate the existing design patterns and strategies used to address these challenges.
3. Develop and propose new design patterns.
4. Compare and contrast advantages and disadvantages of design patterns.
5. Evaluate the design pattern in the context of a selected use-case.

1.3.2 Research Questions

The above-mentioned research aim and objective leads to the following research questions (RQ):

- **RQ1:** Can all safety-critical systems be deployed to the public cloud?
  The latency requirements for different safety-critical cases need to be evaluated to determine whether it is feasible and advisable to run them on the public cloud. By separating safety-critical cases based on their latency requirements, we can better understand which systems may be suitable for public cloud deployment and which may require alternative deployment options.
- **RQ2:** What are the existing fault-tolerance methods in the cloud?
  The fault-tolerance methods available in the cloud are crucial to maintaining system availability and reliability in the event of failures. It is essential to understand these methods and evaluate whether they can be utilized for safety-critical applications. The cloud environment offers a range of fault-tolerance techniques that can be leveraged to enhance system resilience, ensuring that critical applications remain operational and reliable even in the face of failure.
- **RQ3:** How to identify relevant design-pattern?

To identify relevant design patterns, it is crucial to have a deep understanding of the problem space, including the safety-critical use cases and potential failures that may occur on public cloud infrastructure. To accomplish this, it is necessary to use methods that can help identify possible failures and design patterns.

- **RQ4:** Can design pattern be used in various safety-critical systems?
  This requires an investigation into the flexibility and scalability of the proposed patterns and their applicability to different safety-critical domains. By examining the fundamental principles and key features of the design patterns, it is possible to determine whether they can be adapted and extended to other safety-critical systems, thus increasing their utility and value in addressing the challenges of the public cloud environment.

1.4 The Significance of the Study

By exploring the use of on-demand computation power and storage offered by public cloud infrastructure, this study aims to provide a set of design patterns that can be used to mitigate the challenges faced during the development of safety-critical systems with public cloud technologies. The scalability, cost-efficiency, flexibility, and global reach of public cloud infrastructure offer promising possibilities for enhancing the capabilities of safety-critical systems and reducing the risks associated with their operation. As such, this study has implications for industries and businesses that rely on safety-critical systems, as well as for the broader public that stands to benefit from improved safety measures.

Moreover, the study will also identify the limitations and challenges associated with the deployment of safety-critical systems on public cloud infrastructure, as every design pattern beside benefit will have some drawback or disadvantage. This will help in understanding the trade-offs between using traditional approaches and public cloud infrastructure for safety-critical systems.

As a result, this study will contribute to the body of knowledge on designing and architecting public cloud infrastructure that is resilient, redundant, scalable and fault tolerant.

1.5 Limitations of this Study

This study has potential limitations, and it is important to acknowledge it while utilizing its benefits. Hence, there are several areas that will not be explored in this research.

Confidentiality, integrity, and security are important aspects of any system, including those deployed using public cloud technologies. However, these topics will not be addressed in this thesis. Specifically, the focus will be on designing and architecting public cloud infrastructure that is resilient, redundant, scalable, and fault-tolerant, rather than on the security and integrity of the system.

From a real-time system perspective, scheduling algorithms are crucial factors to consider. However, it will also not be addressed in this dissertation. Instead, the study will assume that any type of computation would eventually fail and look for possibilities to recover gracefully from those failures.

Furthermore, it is important to note that this study will focus on general design patterns rather than specific technologies or cloud providers. While this approach allows for broader applicability, it may also limit the depth of analysis and the ability to provide tailored solutions for individual systems. Therefore, further research and refinement of the design patterns may be necessary to address specific technologies or cloud providers.

1.6 Thesis Structure

This section provides an overview of the organization of the remaining chapters in the thesis. Chapter 2 introduces fundamental concepts related to embedded systems, cloud computing, dependability, and fault-tolerance techniques in the cloud. Chapter 3 outlines the research methodology, explaining how design patterns were derived using the Structured What If Technique (SWIFT) methodology from a safety-critical use case. Chapter 4 presents a safety-critical use case from the railway industry along with corresponding design patterns, providing detailed explanations of each pattern's problem, solutions, benefits, drawbacks, and implementation through a use case. In chapter 5, the identified design patterns are evaluated using Fault Tree Analysis (FTA). Finally, in chapter 6, the study concludes by summarizing the main research findings and their relation to the research aim and questions.

2 Literature Review

The aim of this chapter is to synthesize the existing research and connect it with the research aim, which is to create a collection of architectural design patterns that can be used by various safety-critical systems that run on public cloud infrastructure.

2.1.1 Scope

The literature review will cover a variety of topics that are relevant to the development of safety-critical systems in a cloud computing environment. An overview of embedded systems and cloud computing will provide a context for the research. The review will also examine the concept of dependability, fault-forecasting methods and fault tolerance methods for cloud computing systems will be explored.

Considering the vast amount of existing research, following topics will be outside the scope of this review:

- Confidentiality, integrity, and security in the context of dependability
- Scheduling algorithms in the context of real-time systems
- In-depth analysis of run-time cloud offerings, such as Virtual Machines (VM), Containers, Kubernetes, functions, and others
- In-depth analysis of fault-forecasting methods
- Fault-prevention and fault-removal techniques
- A particular cloud technology (e.g., load-balancer) and performance aspect of it

The literature review as well as the thesis will focus on the assumption that, regardless of the type of computation resources that are used, they will eventually fail. Therefore, the aim is to improve fault-tolerance.

2.1.2 Layout

The literature review will start by giving a brief overview of embedded systems and cloud computing, as well as their respective characteristics. Section 2.3 will focus on dependability, which is one of the most important aspects of embedded systems, cloud computing, and safety-

critical systems. The section will introduce the basic taxonomy of dependability and means to attain dependability, including fault-tolerance and fault-forecasting. In section 2.4, cloud computing failures and shortcomings of cloud infrastructure will be explained. In section 2.5 will explore fault-tolerance methods for cloud computing systems. In section 2.6, gaps in current research will be discussed. Finally, the conclusion section will wrap up the literature review and present the key findings. In the following Figure 1, layout of this chapter has been visualized.

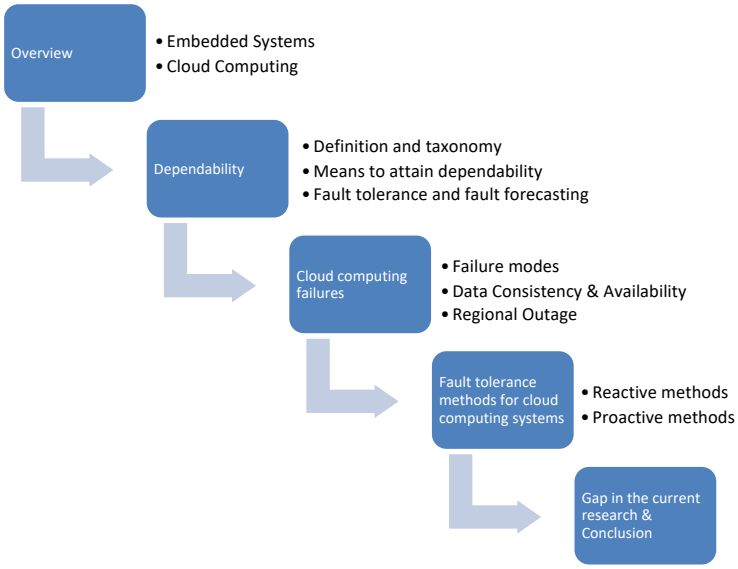

Figure 1: Literature review outline

2.2 Brief Overview of Embedded Systems and Cloud Computing

2.2.1 Brief Overview of Embedded Systems

An embedded systems is an electronic system that is specifically designed and integrated to perform a dedicated task in a well-defined technical environment. These systems are crucial for safety-critical applications, such as medical devices, aerospace systems, and transportation systems, where reliability and safety are of paramount importance (Marwedel, 2021). Embedded systems have several unique characteristics that set them apart from other types of computer systems:

- **Real-time operation:** Embedded systems often need to function in real-time, which requires them to respond to inputs and generate outputs within precise time constraints.

- **Limited resources:** Due to their limited resources, such as processing power, memory, and storage, embedded systems demand meticulous design and optimization to guarantee their proper functionality within the available resources.
- **Specialized environment:** Embedded systems are often designed to operate in specialized and harsh environments, such as industrial settings or transportation systems, which may have specific technical requirements that must be considered during system design and integration.
- **Dedicated function:** Embedded systems are created to serve a predetermined function, which is typically specified during the design phase. This specific design that aims to perform a particular task, ranging from controlling a washing machine to monitoring a spacecraft, is also the system's limitation.
- **Hardware and software integration:** Embedded systems combine hardware and software components to accomplish their intended function. Typically, the hardware and software are closely intertwined and developed together to optimize the performance of the system.

The integration of hardware and software in an embedded system ensures that, it can operate seamlessly and efficiently within its technical environment, reducing the risk of errors and failures. Therefore, the optimization of an embedded system for its intended purpose is essential for achieving high levels of safety and dependability in safety-critical systems(Marwedel, 2021). Dependability is hence a major concern for embedded systems.

2.2.2 Brief Overview of Cloud Computing

The term cloud computing refers to a computing paradigm in which users can access shared computing resources like servers, storage, and applications over a network, enabling users to utilize these resources remotely (Mell et al., 2011).

Cloud providers such as Amazon Web Services, Google Cloud Platform and Microsoft Azure are vendors that offer customers the ability to access and utilize cloud computing resources and services based on their dynamic demand, following a specific business model(Prodan & Ostermann, 2009).

A cloud infrastructure refers to the combination of hardware and software systems that work together to enable the fundamental features of cloud computing, which are:

- **On-demand self-service:** Consumers have the ability to independently provision computing capabilities as required, without the need for direct human interaction with each cloud provider.
- **Broad network access:** Computing capabilities are made available and accessible to users through the network via standard interfaces.
- **Resource pooling:** Cloud service providers adopt a multi-tenant model, combining computing resources to cater to multiple consumers. The provider dynamically allocates physical and virtual resources based on varying consumer demands. Additionally, consumers typically have limited control and knowledge of the exact location of the computing resources provided, except for general regional or geographic information.
- **Rapid elasticity:** Cloud capabilities can be dynamically scaled up or down based on demand, allowing for rapid expansion of resources. This scaling process can be automated, enabling consumers to access computing resources as needed without worrying about capacity limits. From the consumer's perspective, the availability of these capabilities can seem limitless, allowing them to provision resources in any quantity and at any time.
- **Measured service:** Cloud systems effectively manage and optimize resource usage with specialized metering mechanisms for different services like storage, processing, and bandwidth. This enables monitoring, control, and reporting of resource usage, ensuring transparency for both the service provider and consumer (Mell et al., 2011).

These cloud-infrastructure that is offered by cloud providers serviced in three high level model:

- **Infrastructure as a Service (IaaS):** Consumers are empowered to provision vital computing resources like processing, storage, and networks, as well as deploy desired software, including operating systems and applications. However, it is important to note that the provider manages and controls the underlying cloud infrastructure, while the consumer maintains control over operating systems, storage, and deployed applications.
- **Platform as a Service (PaaS):** Consumers are equipped with the ability to deploy their own or acquired applications, utilizing supported programming languages, libraries,

services, and tools on the cloud infrastructure. However, it is important to note that the consumer does not have management or control over the underlying cloud infrastructure, including network, servers, operating systems, or storage. The consumer retains control solely over the deployed applications and potentially the configuration settings for the application-hosting environment.

- **Software as a Service (SaaS):** Consumers are offered the capability to utilize the provider's applications running on a cloud infrastructure, which can be accessed from different client devices through a thin client interface (e.g., web browser) or a program interface. However, it is important to note that Software as a Service (SaaS) is a more restricted model compared to Infrastructure as a Service (IaaS) since it confines consumers to using existing services instead of enabling them to deploy their own applications (Mell et al., 2011; Prodan & Ostermann, 2009).

2.2.2.1 Availability Zones and Regions

Cloud computing resources such as virtual machines, storage, databases, and other services are hosted in geographical locations. These locations are composed of zones and regions. Each region (e.g., Germany West Central) is made up of one or more availability zones, which are distinct physical data centers within a region that are isolated from each other in terms of power, cooling, and network connectivity. The isolation is engineered to provide redundancy and high availability. Furthermore, availability zones in each region are interconnected with high-bandwidth and provide low-latency round-trip (e.g., 2ms). The aim here is redundancy, if one of the zones fails, the others can still provide access to the same resources, given they are designed with replication and redundancy to begin with (*Global Infrastructure Regions & AZs*, 2023; *Regions and Zones | Compute Engine Documentation*, 2023; *What Are Azure Regions and Availability Zones?*, 2023). Figure 2 visualizes the concept of zones and regions.

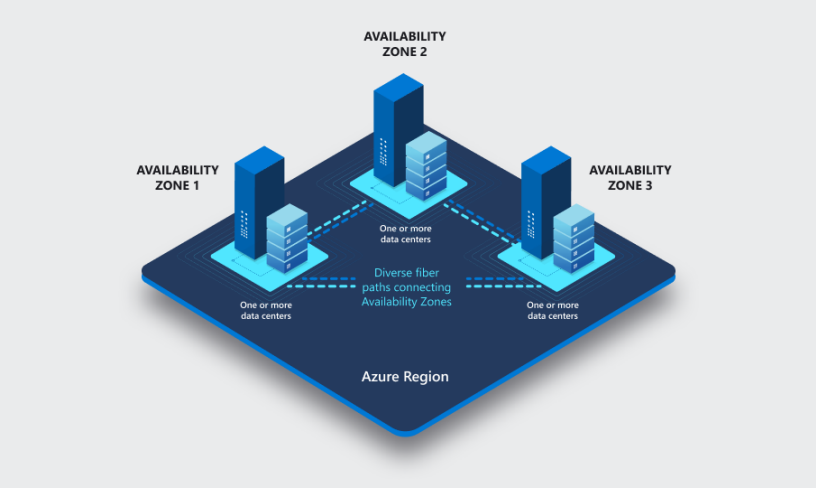

Figure 2: Regions and zones - (What Are Azure Regions and Availability Zones?, 2023)

2.2.2.2 Public, Private and Hybrid Cloud Infrastructure

The definitions introduced in the previous sections refer to public cloud infrastructure, which is owned and operated by companies like Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP). On the other hand, private cloud is dedicated to a single organization and is typically hosted on-premises or in a data center controlled by that organization. While public cloud infrastructure offers numerous benefits, such as scalability and cost-effectiveness, these advantages are not fully realized in private cloud infrastructure due to higher upfront costs and ongoing maintenance expenses associated with infrastructure ownership and management. However, the challenge lies in managing and maintaining a stable and scalable system in the same manner as in the public cloud, which requires extensive knowledge across various fields and highly skilled engineers (Bass et al., 2015).

In recent years, cloud providers have invested significant efforts in developing hybrid cloud management solutions such as Azure Arc, GCP Anthos, and AWS Outpost. These solutions provide capabilities for managing and orchestrating resources, workloads, and applications across hybrid environments, including on-premises data centers. The goal is to achieve consistency, control, and operational efficiency in managing resources and applications across diverse and distributed IT environments (*Azure Arc – Hybrid and Multi-Cloud Management and Solution*, n.d.; *Hybrid Cloud Management with Anthos*, n.d.; *On Premises Private Cloud - AWS Outposts - AWS*, n.d.).

Although this thesis primarily focuses on public cloud infrastructure, it is worth noting that certain safety-critical systems may not be able to run on public cloud infrastructure due to regulatory requirements. In such cases, these hybrid cloud management solutions serve as interesting alternatives, allowing private entities to maintain control over their hardware infrastructure while leveraging a management layer similar to that of the public cloud. The regulatory aspects of this issue are briefly addressed in the next section under "Governance and Security".

2.2.3 Integration of Embedded Systems with Cloud Computing

Embedded systems are ubiquitous in modern society, powering everything from smart home devices like thermostats and security cameras to complex machinery like medical devices and automotive systems. Differences in the type of systems, also means that cloud computing can be integrated with embedded system in different ways, from simply leveraging cloud for data storage and analytics to the extend having complete control loops being deployed and executed in the cloud. The only exception to this integration is in processes that involve sensors and actuators that sense and control the physical environment (Hallmans et al., 2015).

Hallmans et al., (2015) has identified several 5 major challenges, running embedded systems on the cloud:

- **Timing:** Hard real-time systems have strict timing requirements that can be difficult to meet on a cloud server due to resource sharing and potential interference from other applications. While less extreme timing requirements may be feasible for cloud deployments, the challenges of meeting strict timing requirements must be considered.
- **Communication:** Even though ethernet-based communication allows for cloud-based applications to communicate with sensors/actuators, using different real-time network communication protocols poses a challenge that requires embedding data via a gateway without compromising performance. Adding communication between a local sensor and a remote cloud system increases the chances of failure if no redundancy is used, and loss of communication with the cloud must be managed similarly to how communication bus failures are handled in current systems.

- **Redundancy/control-/protection-system:** Cloud systems used in industry must provide high availability through redundancy and fast switching between systems. Other requirements include having different physical instances for control loops and protective functions, which work together to supervise control actions.
- **Governance and security:** Moving the control functionality of a power station to a cloud owned by a third-party organization raises security and governance concerns related to physical security, human resource security, business continuity, disaster recovery, identification and access management, encryption, and government regulations. While similar requirements exist for other industries, the critical infrastructure status of power stations requires extra caution.
- **Safety certification:** Certifying safety applications running on the cloud presents challenges due to concerns about hardware limitations and communication issues for certified processes, whether and how safety-critical applications running in the public cloud can be certified has to be clarify.

The timing or scheduling of tasks is indeed one of the biggest challenges of moving control loops from embedded systems to cloud computing. In fact, there is an argument, that current cloud computing system, as it may be useful to enterprise computing, it is not fit for safety-critical systems, because integrated functions cannot guarantee absolute performance as well as resource use cannot be determined in advance(Jakovljevic et al., 2014).

To give a hypothetical but precise example, let us consider an airbag system in modern vehicles. These systems are considered safety-critical due to their precise timing requirements. Inflation of an airbag too early or too late can cause harm to passengers rather than protect them. Typically, an airbag system needs approximately 20ms to detect a crash and another 20-30ms to inflate the airbag (Birdsong et al., 2006). The time from the moment of the crash to complete inflation of the airbag is approximately 50-60ms, depending on the vehicle's velocity. Most of these 60ms is used by sensing and actuating, hence running a control loop of an airbag in the cloud would be impossible just by the sheer fact of latency.

The question that arises is whether cloud infrastructure can support safety-critical applications and what would be the anticipated latency. While numerous factors can affect latency, including processing power, programming language, computing algorithm, and network setup, the most

significant factor in this case is the physical distance from the cloud infrastructure. Figure 3, displays an analysis of latency in terms of round-trip time (RTT) of major cloud computing vendors, showing the time it takes for data to travel from one point of origin to various locations. The analysis is conducted via PingMesh tool using TCP and HTTP(S) protocols. The lowest RTT value(28ms), in the Middle Eastern region is due to the proximity to Muscat(point-of-origin), as shown in the figure. The latency analysis reveals that South Africa and Sao Paulo have higher latencies due to their expected network distance of 7158 and 11650 miles (about 18748.86 km), respectively, resulting in slower access. The network distance to Bahrain and Dubai is around 17 times less than the intercontinental distance to South Africa and Sao Paulo, which emphasizes the impact of geographical distance on round trip latency. The authors of the study argue that latency under 100ms is considered acceptable for normal use cases, but for time-critical applications, a lower value might be necessary. (Zen et al., 2022).

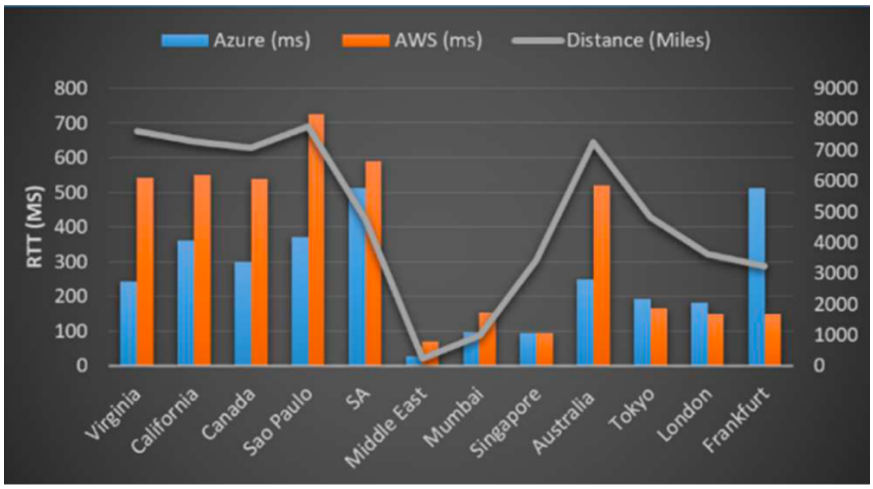

Figure 3: Latency vs Distance - (Zen et al., 2022)

Given current latency measurements, it should be possible to run some safety-critical systems that have timing requirements around a couple of seconds on cloud infrastructure. Although at this stage, none of the cloud providers offer specialized hardware or scheduling algorithms to deliver timely results, with enough processing power, redundancy, and a well-designed system, achieving timely delivery should be possible.

2.3 Dependability

In the context of safety-critical systems, dependability is a fundamental property that refers to the system's capability to provide its intended functionality in a reliable, consistent, and timely manner, with the aim of ensuring that it does not pose any unacceptable risks to people, the environment, or property.

Dependability, it is defined as “ability to avoid service failures that are more frequent and more severe than acceptable”(Avizienis et al., 2004). Figure 4, visualizes the taxonomy of dependability.

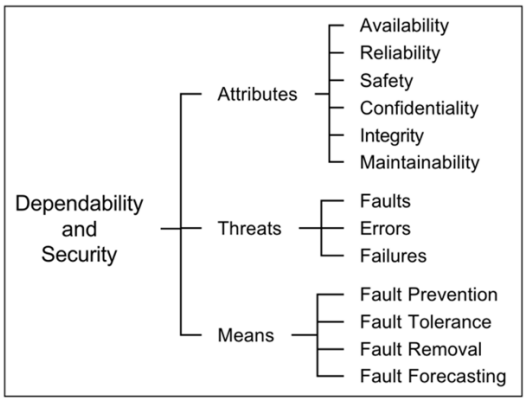

Figure 4: The dependability and security tree (Avizienis et al., 2004).

2.3.1 Dependability Attributes

Dependability attributes are characteristic or properties that a dependable system or process must have to fulfill its intended function. One can utilize these attributes as a structure for assessing the dependability of a system or process. Dependability encompasses following quality attributes (Avizienis et al., 2004):

- **Availability**: System's ability to function correctly and be ready for use when needed.
- **Reliability**: System's ability to consistently function correctly over time without interruption or failure.
- **Safety**: System's ability to prevent harm to users and the environment, by avoiding catastrophic consequences.

- **Confidentiality**: System's capacity to safeguard sensitive information and prevent unauthorized disclosure.
- **Integrity**: System's ability to maintain its intended functionality and data quality, by preventing improper system alterations.
- **Maintainability**: System's ability to be easily maintained and repaired, allowing for modifications or bug fixes without disruption to the system's operation.

2.3.2 Dependability Threats

Threats are defined as events or conditions that can lead to system or process failure or deviation from the intended behavior. Various sources, including hardware or software defects, human error, environmental conditions, and malicious attacks, can give rise to these threats. It is crucial to identify and mitigate them to ensure that a system or process remains dependable and can execute its intended function.

A **fault** is a defect or flaw in a system, which may or may not lead to an error or failure. For example, a coding error or a design flaw in a software system is a fault.

An **error** occurs when the actual behavior of a system inside its boundary does not match its intended behavior. This can happen when the system enters an unexpected state due to the activation of a fault, leading to a discrepancy between what was intended and what actually happened.

A **failure** occurs when a system behaves contrary to its intended specifications. A failure happens when an error is not resolved correctly, either because it was not handled properly, or the system failed to recover from the error. However, an error does not always result in a failure. If an error occurs in a system, it may throw an exception. However, the exception can be caught and managed using fault tolerance techniques to ensure that the system continues to operate as intended and according to its specification. This means that the system can recover from the error and continue functioning without causing a failure.

Fault, error, and failure are tightly coupled. The activation of a fault causes an error, which, if not handled properly, can lead to a failure, and render a system unusable(Avizienis et al., 2004).

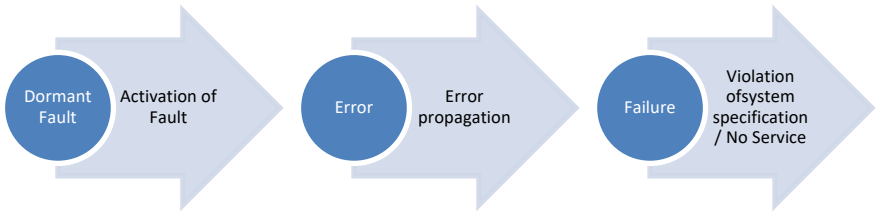

Figure 5: Relationship of fault, error, and failure

2.3.3 Means of Achieving Dependability

After grasping the fault-error-failure chain that was explained earlier, the focus shifts to ensuring that a system or process functions as intended without any unexpected failures, errors, or malfunctions, and improving its dependability. To achieve the attributes of dependability and security, four means have been developed(Avizienis et al., 2004):

- **Fault prevention:** A set of techniques used to prevent the occurrence or introduction of faults into a system.
- **Fault tolerance:** The ability of a system or component to prevent service failures even in the presence of faults.
- **Fault removal:** The process of identifying and eliminating faults to minimize their quantity and impact in a system.
- **Fault forecasting:** A proactive approach to predict the current and future occurrence of faults, along with their potential consequences.

In this thesis, fault forecasting will be used to identify potential failures and derive design patterns that are fault tolerant.

2.3.3.1 Fault Tolerance

Fault tolerance is a technique used to ensure that a system can continue to operate correctly in the event of a failure. The goal of fault tolerance is to avoid failures and maintain system availability and reliability. There are several ways to achieve this:

- **Error detection:** This involves utilizing different mechanisms, such as redundancy, checksums, and error-correcting codes, to identify errors within the system. Redundancy involves duplication of critical components increase reliability or fail-over.

Checksums involves calculating a numerical value from the data to detect errors in data transmission or storage, and then comparing it to a known value.

- **System recovery:** This involves taking necessary actions to restore the system to its normal operation after a failure has occurred. There are different methods including restarting failed components, switching to backup systems, or initiating a failover to a redundant system, are used in this process. This process can be either manual or automatic, depending on the system and the nature of the failure.

Fault tolerance is an essential aspect of system design, particularly in critical systems where downtime can have severe consequences (Avizienis et al., 2004). Therefore, in section 2.4 cloud computing failures as well as the fault tolerance techniques that are used in the cloud infrastructure will be examined in detail.

2.3.3.2 Fault Forecasting & Techniques

Fault forecasting is used ensure a system's dependability by identifying and predicting potential faults or failures in systems or processes before they occur. Hence, fault forecasting techniques refer to a set of methodologies and tools that are used to predict potential faults or failures in systems or processes. Fault forecasting approaches can be categorized into two types: bottom-up (inductive) approaches, which involve starting with a failure and then identifying related consequences through inductive reasoning, and top-down (deductive) approaches, which involve starting with a failure or undesired effect and identifying related causes through deductive reasoning (Adler, 2019).

There are various techniques, such as Fault Tree Analysis (FTA), Failure mode and effects analysis (FMEA), Hazard and Operability Study (HAZOP), The Structured What-if Technique (SWIFT), each of which can be used to systematically identify and analyze potential faults, failures, and hazards in a system, allowing for effective risk management and the implementation of appropriate preventive or corrective measures.

Elaborating on each of these methods is beyond the scope of this thesis. As this thesis will use SWIFT for deriving design patterns from a safety-critical use case, this method will be explained briefly, and the justification for why this method has been selected will be explained in the next chapter.

2.3.3.3 Structured What If Technique (SWIFT)

The Structured What-If Technique (SWIFT) is a technique for identifying system-level risks and hazards, which involves structured brainstorming using predefined headings or guidewords (e.g., timing, human-factors, environment, etc.), combined with prompts beginning with phrases such as "*what if...*" or "*how could...*" to thoroughly examine risks and hazards. It is a tool that is both simple and effective, and it can be particularly useful when applied early in the design process and throughout the life cycle of a system. The SWIFT process involves preparing guidewords, assembling a team, providing background information, defining the purpose, describing the system, identifying risks and hazards using the SWIFT technique, assessing the risks, proposing risk control actions, reviewing the process, producing an overview document, and conducting additional risk assessments if necessary. The method is used widely in chemical, petrochemical, energy, manufacturing, high-tech, food processing, transportation, and healthcare. (Card et al., 2012; Lyon & Popov, 2021).

According to a systematic literature review conducted by Card et al. in 2012 in the healthcare sector, it was concluded that healthcare workers find SWIFT easy to learn, easy to use, and credible, as well as less time-consuming than FMEA. Compared to FMEA, producing overlapping but different results(Card et al., 2012). It is worth mentioning that the authors acknowledge that the evidence available to them was limited.

2.4 Cloud Computing Failures

Despite cloud providers offering a high level of service availability, failures still occur. The high availability being offered is for the large segment of the service, but individual hardware failures, software failures, and configuration failures still take place. Bass et al. (2015) reported that AWS released data indicating that in a datacenter with approximately 64,000 servers, each equipped with two disks, an average of more than five servers and 17 disks fail each day. The authors also mentioned similar numbers in the GCP in a single datacenter, including thousands of hard disk failures per year, connectivity loss, overheating, router failures, and more (Bass et al., 2015). In this section, challenges, issues, and fault-tolerance techniques are identified, which are imperative to run safety-critical applications on cloud infrastructure.

2.4.1 Cloud Computing: Failure Modes

As mentioned in the previous section, cloud computing, despite offering high availability, is susceptible to failures. Mesbahi et al. (2018) investigated possible failure modes and classified them into six categories, the following list summarizes their findings. Each category also contains the failure mode, for instance hardware failures are divided between “hardware component failures” and “network failure” modes:

- **Software Failures:** Cloud tasks and VM hypervisors may have software faults that cause system/application failures. Databases can be vulnerable to hardware or software failures, which can lead to the loss of data.
- **Hardware Failures:** Computing resources, like storage devices, processing elements, and memory, can experience hardware failures. Also, network failure can happen during cloud tasks accessing remote data sources.
- **Cloud Management System Failures:** Cloud management system uses a combination of software and technologies to handle cloud environments, manage pools of resources, monitor and track resource usage, and more. Typical failures here include overflow failure, timeout failure, missing data resource, and missing computing resource.
- **Security Failures:** Cloud service failures can be caused by software security breaches, where attackers gain unauthorized access to customer information through cloud-based software. Another common reason for security failures is miscalculating the security requirements for a comprehensive security policy. While some of these failures can be attributed to the provider, research suggests that the majority of cloud security failures are caused by their customer.
- **Human Operational Faults:** The failure is caused by human error during the operation or configuration of the system, which can impact the cloud system. Misconfiguration of the network or other underlying systems can also bring down the entire cloud system.
- **Environmental Failures:** Environmental disasters such as floods, power outages, and fires can interrupt service provision of a cloud system. They are outside the control of the service provider and can cause large-scale disruptions. Other issues such as failure in the air-conditioning system of a cloud datacenter can also lead to service failures (Mesbahi et al., 2018).

All these failure modes can impact the availability and reliability of the systems running on the public cloud. However, being aware of potential failure modes can help to create guards against these failures.

2.4.2 Cloud Computing: Data Consistency & Availability

Data storage technologies are one of the most critical parts of any application. There has been a lot of research and advancement in the field, and cloud providers offer different data storage technologies tailored to various use cases. Given the possible failures that can occur, data is constantly replicated. Cloud providers offer geo-replication, which allows data to be written in one geographical region and eventually read in another region with low latency. However, basic physical limitations still exist. In other words, although it is much easier to use a database that has a geo-replication feature, it is important to understand the consistency aspect of it.

In the context of data storage technologies and cloud providers, it is essential to consider the CAP theorem, which states that consistency, availability, and partition tolerance cannot be simultaneously achieved in a distributed system(Gilbert & Lynch, 2012). Achieving optimal geo-replication involves finding the right equilibrium between consistency and availability.

Consistency guarantees that regardless of the node a client connects to, all clients will observe the same data simultaneously. With geo-replication, achieving strong consistency can be challenging, especially when dealing with high-latency network connections between data centers. In some cases, data inconsistency may occur when multiple clients try to access and update the same data simultaneously. To mitigate this risk, some cloud providers offer options for configuring the consistency level of geo-replicated data.

In summary, while geo-replication is a convenient feature for accessing data across different regions with low latency, it is important to consider the consistency aspect of it. Cloud providers may offer different consistency levels for geo-replicated data, and it is up to the application developers to choose the appropriate level that balances consistency and availability based on their specific use case.

2.4.3 Cloud Computing: Regional Outage

As it was explained in the section Brief Overview of Cloud Computing, cloud providers offer regions and zones that provide redundancy mechanisms, if one of the zones fails, the others can still provide access to the same computing resources and data that has been replicated across zones. However, in context of safety-critical system this may not be sufficient, hence this section would argue, why single region design is not appropriate for safety-critical system, and it would even take a step forward arguing it might even be appropriate to have multi-cloud strategy.

Within a geographical cloud region, multiple zones exist, with each zone housing one or more data centers. However, the probability of regional outage or scarce computational resource occurring in a particular region is still possible. Considering the following cases:

- **Human Operational Faults**: An outage in cloud computing, as mentioned in section 2.4.1, can occur even when the underlying hardware and software are still functional, due to other human errors.
- **Capacity Issues**: Limited available compute resources can limit scalability and have an impact on the availability of a running system.
- **Environmental factors**: These factors, such as floods, earthquakes, and cooling system failures, cannot be foreseen, but there is a possibility that they might happen.
- **Cascading Failure**: Failure in the cooling system of a single availability zone can cascade other failures, such as scarcity of resources. For instance, assuming that a single availability zone suddenly becomes unavailable because of a cooling system failure, the remaining workload has to be migrated to other zones. However, this strategy assumes that the remaining zones have enough capacity to take over the workload of the failed zone.

Hence, the possibility of regional failure is very likely, and therefore a single-region design for a safety-critical system is not sufficient. Furthermore, if there is a regional outage, this can trigger the disaster recovery strategy of most cloud consumers. The natural tendency of most cloud consumers would be to choose the closest geographical region; however, this clustering can lead to scarce resources in nearby regions. Therefore, a multi-cloud strategy, which will be explained in depth in chapter 4, should be considered for safety-critical systems.

2.5 Cloud Computing Fault Tolerance Methods

In the context of cloud computing, fault tolerance approaches are divided into two primary categories: reactive and proactive(Agarwal & Sharma, 2015; Amin et al., 2015; Ataallah et al., 2015; Hosseini & Arani, 2015; Mittal & Agarwal, 2015; Prathiba & Sowvarnica, 2017). There are multiple methodologies in the literature that have been implemented to achieve fault tolerance based on these approaches. Although adaptive fault tolerance and machine learning fault tolerance are suggested techniques as well, proactive, and reactive are the primary fault tolerance methods.

2.5.1 Reactive Methods

In the context of reactive fault-tolerant techniques, the focus is primarily on restoring the system to its normal state after a fault occurs. To achieve this, the system state is continuously saved and used during the recovery process. Replication, checkpointing, and restarting are some of the fundamental techniques utilized for this purpose.

Algorithm/Technique	Description
Checkpoint/Restart	This method relies on continuously saving system state, in the event of failure, the task is resumed from the last persisted checkpoint(Amin et al., 2015; Ataallah et al., 2015; Hosseini & Arani, 2015; Mittal & Agarwal, 2015; Mukwevho & Celik, 2021; Prathiba & Sowvarnica, 2017; Rehman et al., 2022). (Mukwevho & Celik, 2021; Patra et al., 2013)
Replication	This method relies on various components being copied and deployed simultaneously across different resources, aim is to increase availability and execution of tasks (Amin et al., 2015; Ataallah et al., 2015; Hosseini & Arani, 2015; Mittal & Agarwal, 2015; Mukwevho & Celik, 2021; Prathiba & Sowvarnica, 2017; Rehman et al., 2022).
Job Migration/Task Re-submission	This method works by migrating/re-submitting the failed task to either same computation resource or to a different one (Amin et al., 2015; Ataallah et al., 2015; Hosseini & Arani, 2015; Mittal & Agarwal, 2015; Mukwevho & Celik, 2021; Patra et al., 2013; Prathiba & Sowvarnica, 2017; Rehman et al., 2022)

Retry	This method works by retrying the failed request on the same resource. Although it is a simple method, it is a very effective one(Amin et al., 2015; Hosseini & Arani, 2015; Mukwevho & Celik, 2021; Patra et al., 2013; Prathiba & Sowvarnica, 2017; Rehman et al., 2022).
Timing Check	This method works by components resetting a timer. When a component or system fails to reset the timer, it is the indication of a fault(Hosseini & Arani, 2015; Patra et al., 2013) It is also known as watchdog timer ("Watchdog Timer," 2023).
N-Version and Recovery Block	This method works by creating redundant copies of the software in different ways to reduce the probability of the similar faults in two or more copies. N-Version programming is based on the idea that multiple independent teams develop identical software, and these redundant copies of the software run concurrently. In contrast, the Recovery Block approach involves creating multiple copies of the software using different algorithms, and these redundant copies are not executed concurrently but rather sequentially until acceptance test is passed(Mukwevho & Celik, 2021).
Rescue Workflow	This method works by facilitating the workflow's continuity despite task failures, until it becomes impossible to continue without addressing the failed task. The idea behind the method is to persist the state of failed and succeeded task, and at next execution re-submit the failed workflow but only attempt to compute the failed task, saving computation resources(Hernandez & Cole, 2007; Hosseini & Arani, 2015; Mukwevho & Celik, 2021; Patra et al., 2013; Prathiba & Sowvarnica, 2017)

Table 1: Reactive fault tolerance methods

2.5.2 Proactive Methods

Proactive fault-tolerant techniques focus on predicting and preventing faults before they occur. The system is constantly monitored, and failure prediction algorithms are used to assess its status, enabling necessary actions to be taken to prevent failures. In cloud systems running on virtualized environments, fault management techniques rely on migration and pause/un-pause functionality provided by the virtual platform. Key techniques used for proactive fault tolerance include software rejuvenation, self-healing, and preemptive migration.

Algorithm/Technique	**Description**

Load Balancing	This method relies on distribution of the workload across components that perform the same function. It effectively works together with replication method that was mentioned above. It is one of the most important methods of fault-tolerance in the context of the cloud computing(Mukwevho & Celik, 2021; Rehman et al., 2022).
Preemptive Migration	This method relies on a feedback control mechanism to monitor the state of system components and remove or suspend components that are likely to fail. By implementing this approach, any adverse effects on the overall system performance can be effectively minimized. (Amin et al., 2015; Ataallah et al., 2015; Hosseini & Arani, 2015; Mukwevho & Celik, 2021; Patra et al., 2013; Prathiba & Sowvarnica, 2017; Rehman et al., 2022)
Software Rejuvenation	This method relies on restoration of the initial state of the software by simply implementing periodic graceful shutdown and restart of the component (Amin et al., 2015; Ataallah et al., 2015; Hosseini & Arani, 2015; Mukwevho & Celik, 2021; Patra et al., 2013; Prathiba & Sowvarnica, 2017; Rehman et al., 2022)
Self-healing	This method relies on feature of the system that enables automatic detection, diagnosis, and repair of both software and hardware faults without human intervention(Amin et al., 2015; Hosseini & Arani, 2015; Mittal & Agarwal, 2015; Mukwevho & Celik, 2021; Prathiba & Sowvarnica, 2017; Rehman et al., 2022).
SGuard	This method relies on saving checkpoint asynchronously while services are still running, failed services are rolled back to latest error-free state and recovered. Checkpoints are saved in a distributed storage, as a result services are replicated. Hence the method can be used in distributed stream processing engines and offers better performance and reliability(Amin et al., 2015; Hosseini & Arani, 2015; Kwon et al., 2008; Mittal & Agarwal, 2015; Mukwevho & Celik, 2021; Patra et al., 2013; Rehman et al., 2022).

Table 2: Proactive fault tolerance methods

Many methods that were described above can be used in combination to increase availability and reliability of the system. For instance, a system can be composed of many components; replication methods can be used to create a copy of deployed software to guard against hardware failures. The utilization of a load balancer allows for the distribution of computational tasks across multiple instances of the running software. Furthermore, assuming the system has a clearly defined interface, the client can use the retry method to re-request a piece of data from

the system. Therefore, most of these methods should be seen as complimentary to increase the availability and reliability of the system.

2.6 Gap in the Current Research

There are several gaps that have been identified in the current research:

- Focus on scheduling algorithms to manage timing constraints, rather than utilizing the available cloud infrastructure.
- Limited number of practical safety-critical applications running on public cloud.
- Lack of clearly identified methodologies to manage possible failures.
- Overall available research in the area of safety-critical systems running on cloud infrastructure.

There are naturally many reasons why there is currently such a gap in the existing research. For instance, cloud providers at this stage may not have any incentive to invest in customized hardware and scheduling algorithms for safety-critical systems. This lack of interest may be due to the belief that the added cost and effort of such specialized systems outweigh the benefits, as simply providing cloud technologies on commodity hardware is still a very profitable business. Therefore, relying solely on scheduling algorithms as a solution at this stage is not sufficient. Furthermore, it should be possible to overcome the lack of proper scheduling algorithms and hardware via redundancy and available extra computation power.

Despite the growing popularity of public cloud infrastructure, it appears that there has not been enough research focusing on running safety-critical systems in the public cloud environment. As a result, there is a pressing need for more research and development in this area. By identifying and addressing the challenges associated with running safety-critical systems on the public cloud, researchers may be able to unlock the full potential of cloud infrastructure for critical applications in fields such as healthcare, transportation, and energy.

Overall, there is much work to be done in this area, and the development of new techniques and technologies will be crucial in ensuring the safety and reliability of critical systems in the public cloud. By pursuing these research efforts, we can pave the way for a safer, more efficient, and more cost-effective approach to safety-critical systems in the cloud.

2.7 Conclusion

The literature review provides valuable insights into the existing practices and techniques used in dependability engineering and fault tolerance for safety-critical systems in the public cloud. It is clear that there are a number of well-established practices that can be applied to this domain, such as fault forecasting techniques including FMEA, SWIFT, and HAZOP. These techniques help to identify potential faults and their consequences, which can then inform the design of more robust and fault-tolerant systems.

Moreover, fault tolerance techniques are also an important consideration for ensuring the reliability and safety of critical systems in the public cloud. These techniques include hardware redundancy, software redundancy, checkpointing, and error correction codes. These techniques can help to detect and recover from faults in hardware, software, or data, and enable systems to continue to operate even in the presence of faults or failures.

However, while these techniques are important, there are still challenges to overcome when it comes to ensuring the reliability and safety of critical systems in the public cloud. One of the main challenges is the reliance on commodity hardware and shared infrastructure, which may not meet the strict requirements of safety-critical systems. Additionally, scheduling and hardware optimization have been proposed as solutions, but there is still limited exploration of other potential strategies.

To address these challenges, further research is needed in several areas. One promising approach is to explore alternative solutions to scheduling and hardware optimization, such as leveraging redundancy methods and processing power. This approach could help to ensure that safety-critical systems with timing requirements of several seconds or more can still be supported, even without specialized hardware and scheduling algorithms.

Another important area of research is to develop new techniques and technologies to ensure the safety and reliability of critical systems in the public cloud. This could include the use of machine learning or other data-driven approaches to predict faults and adapt system behavior accordingly. Additionally, the impact of shared infrastructure and commodity hardware on safety-critical systems in public cloud infrastructure must be studied more extensively.

Finally, case-studies and simulations can be used to test the effectiveness of different solutions and strategies for safety-critical systems in the public cloud. This can help to identify areas where improvements can be made, as well as provide a more detailed understanding of the trade-offs between different approaches.

Overall, the literature review highlights that there is still much work to be done in ensuring the safety and reliability of critical systems in the public cloud. However, by pursuing further research efforts in these areas, we can develop new and more effective strategies for achieving timely results and ensuring the safety-critical systems that are running on public cloud infrastructure.

3 Deriving Design Patterns from Use Cases Using the SWIFT: Methodology and Data Collection

This section describes the purpose and procedure of data collection using the proposed SWIFT-based methodology for deriving design patterns from a selected use case. The process involved identifying the use case and relevant input parameters, applying SWIFT to systematically explore what-if scenarios, and analyzing the results to identify patterns in the system's behavior. The collected data were used to answer research question, ***RQ3***, and evaluate the proposed methodology's effectiveness in deriving design patterns from use cases.

3.1 Introduction

Design patterns play a crucial role in designing software by offering reusable solutions to recurring design challenges encountered during the development process, based on past experience, thereby enhancing flexibility, reusability, and efficiency in design decision-making (Gamma et al., 1995). However, identifying and selecting appropriate design patterns heavily depends on experience, observation, and analysis of the given problem, as there is currently no standardized method for creating design patterns. Moreover, the problem for which a design pattern is being considered must be a recurring one in similar systems, as design patterns are intended to provide reusable solutions that can be applied to subsequent systems.

One approach to addressing this challenge is to derive design patterns from use cases. Use cases describe system behavior and capture interactions between users and the system. Therefore, the primary purpose of use cases is to define and document the functional requirements of a software system. Deriving design patterns from use cases can help ensure that the patterns closely align with the system's requirements and similar use cases.

In this research, a methodology is proposed for deriving design patterns from use cases using the Structured What-If Technique (SWIFT). SWIFT methodology which was introduced in section 2.3.3.3, is a technique for identifying system-level risks and hazards, which involves structured brainstorming using predefined guidewords. By applying SWIFT to use cases, patterns in the system's behavior regarding risks or failures can be identified, which can then be used to derive design patterns.

The proposed methodology involves the following steps:

- Identifying a safety-critical use case based on real-world scenarios.
- Setting up the initial context by running the safety-critical use case on cloud infrastructure.
- Identifying guide words to be used in SWIFT analysis.
- Applying SWIFT to the use case to systematically explore what-if scenarios.
- Analyzing the results of the SWIFT analysis to identify patterns in the system's behavior.
- Deriving design patterns from the identified behavior patterns of the system.

The proposed methodology will be applied to a use case to demonstrate its effectiveness in deriving design patterns. The proposed methodology has the potential to enhance the efficiency and effectiveness of the design pattern selection process, leading to an improved quality model for the software product, particularly in terms of reliability, as per ISO/IEC 25010.

3.2 Use Case Selection

For the purpose of deriving design patterns, a number of real-world scenarios have been examined, such as an autonomous fleet of robots in a warehouse, automated valet parking, positive train control, and moving block in railway signaling. As a result, moving block in railway signaling has been selected as the use case for this thesis. Several factors favored this use case over the others, these are:

- Moving-block software could be treated as a black box because it has few input parameters and produces precise output values.
- Timing constraints are not very tight, as computation occurs every several seconds, which can be easily handled by the cloud infrastructure.
- The railway industry has traditionally relied on dedicated and specialized hardware on the track to grant authority of movement to trains. By moving some of this operation to the cloud, it is possible to reduce overall equipment, cost, and maintenance effort.
- Necessity to access comprehensive information about all trains at a centralized location.

In summary, a real-world scenario was selected to derive design patterns, resulting in the selection of the moving block in railway signaling as the use case due to its ability to be treated as a black box with precise outputs, low timing constraints, and the potential to reduce equipment, cost, and maintenance through cloud infrastructure. Detailed explanation of the use case would be presented in the next chapter.

3.3 What-If Context

As risk analysis in a brainstorming session can become complicated due to the large number of factors involved, it is important to set the context of the analysis to ensure the correct identification of potential risks. In this study, the context was reduced to:

- Any potential risk associated with cloud infrastructure. Failure modes that were identified in the literature review were used as bases.
- Hypothetically, it is assumed that all software components of the moving-block system have been accurately developed, well-tested, and are functioning correctly as intended.

This approach ensures that efforts are focused on identifying design patterns related to safety-critical systems running on cloud infrastructure, which can be re-used in different use-cases.

3.4 Identification of SWIFT Guidewords

In SWIFT, guidewords refer to a set of words that are used to systematically generate potential deviations or failures from a given scenario or situation. The purpose of using these guidewords is to facilitate brainstorming of potential causes and consequences of deviations and to steer the analysis towards achieving a more holistic understanding of the system under examination (Card et al., 2012; Lyon & Popov, 2021).

In the context of running safety-critical applications on public cloud, several sources have been used to identify the guidelines. These are:

- Dependability quality attributes from section 2.3.1 have been used as input.
- Failure modes of cloud computing from section 2.4.1 have been used input.
- Cloud architecture frameworks that are advised by three major cloud providers has been used as input 2.3.1.

The following Table 3 shows an excerpt of the list of identified SWIFT guidewords. All the identified guidewords can be referred to in Appendix 8.1

Categories	Guidewords	Description
Availability	• Uptime • SLA (Service Level Agreement) • Load Balancing • Auto Scaling • Disaster Recovery	This guideword refers to the ability of the application to always remain accessible and functional to users. Potential issues related to availability include downtime, service disruptions, and network connectivity issues.
Performance	• Resource Utilization • Application Tuning • Query Optimization • Caching • Indexing	This guideword refers to the speed, responsiveness, and overall performance of the application. Potential performance issues include slow response times, resource bottlenecks, and inefficient code, algorithm, or inability to handle large volumes of traffic.
Data-Consistency	• Replication • Consensus • Atomicity • Isolation • Durability	This guideword refers to the accuracy and integrity of data that is stored and accessed by the application. Potential issues related to data consistency include data corruption, data duplication, and inconsistencies between different data sources.

Table 3: Excerpt of identified SWIFT guidewords.

3.5 Data Collection and Analysis: Generating What-If Questions

Given the use case, context, and guidewords, what-if sessions were conducted. The guidewords were used in combination with “what-if” and “how-could” prompts to generate questions. This process was iterative, as applying design patterns changes the system and a new set of questions may need to be asked to verify whether the system under consideration has any new potential risks. The following Figure 6 visualizes this process.

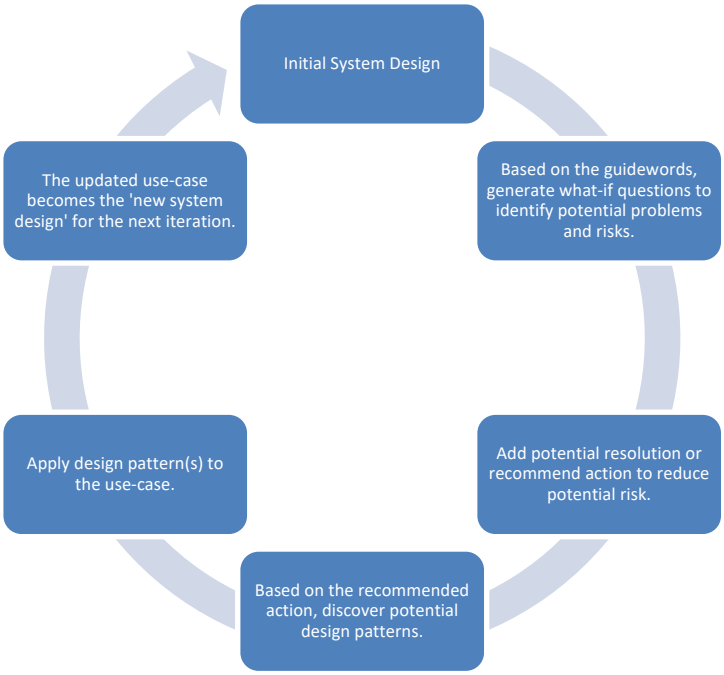

Figure 6: SWIFT process

At the beginning, the first system design was intentionally very rudimentary. Components did not have any redundancy of any sort, and the system was treated as a single black box. The aim of the process was to start with a rudimentary system to identify all possible design patterns, without making any assumptions about the system under consideration. After each iteration, the system would evolve, and the applied design pattern(s) would become the 'initial system design'.

Based on the context, guidewords, and system under consideration, potential risks were identified using what-if/how-could questions. For instance:

- What if the compute system running the software goes down unexpectedly?
- What if the system experiences a sudden increase in traffic, causing it to slow down or crash?

Next, possible recommendations were identified to reduce the potential risks, such as "*introduce redundancy*" or "*rely on auto-scaling*". The following Table 4 shows an excerpt of the identified "*what if...*" questions as well as the recommended actions. All identified questions can be referred to in Appendix 8.2.

"What if…" / "How could…" Questions	Recommended Action	Proposed Design Pattern
What if there is a lack of isolation between different software components?	Enforce logical and physical separation between different parts of the systems.	Critical Enclave
What if a disaster or service disruption occurs, impacting the availability of critical systems or data?	Implement redundant infrastructure.	Multi-Region Deployment, Geo-Replication
What if the application's response time is slow or inconsistent?	Implement timeout and retry mechanism to reduce the propagation of potential failures.	API Gateway

Table 4: Excerpt of "what if..." questions

Next step was to identify potential design patterns. Design patterns were documented with context and problem it addresses, solution, benefits, and drawbacks as well as an example. Use case was utilized to demonstrate an example of the design pattern. In some cases, design pattern has become part of the new system design. Finally, the new system design was used as an input to another round of “what if…” session, which started from the start.

3.6 Methodological Limitation

The limitations of this study include the availability and accessibility of use cases that can be used to derive design patterns. Additionally, the use of deductive reasoning to derive design patterns may limit the generality of the patterns to specific use cases, although this issue has been considered throughout this thesis. Furthermore, the SWIFT, being a high-level investigation tool that can identify hazards without the need for a detailed review of low-level processes and equipment subcomponents, enables fast iteration of potential risks. Furthermore, following the completion of the process, a detailed evaluation such as FMEA would be necessary to ensure that no potential issues have been overlooked.

3.7 Conclusion

This chapter presents the methodology adopted for the research. The use of structured what-if technique will be used to derive design patterns from a use case. The primary data collection method will be supplemented with a literature review. The data will be analyzed using deductive reasoning, and the validity and reliability of the study will be ensured. The limitations of the study have also been identified.

4 Architecture of Safety-Critical Applications Running in the Public Cloud

Designing safety-critical systems that can run on public cloud infrastructure is a complex task that demands a deep understanding of both the system's requirements and the intricacies of cloud infrastructure, as well as a meticulously planned strategy. In the scope of this thesis, the aim was to produce a set of design patterns that can be applied to various safety-critical systems. Therefore, in this section, a use-case will be introduced, and the identified design patterns will be applied to the use-case iteratively.

4.1 Use Case – Moving Block Interlocking

4.1.1 The European Rail Traffic Management System

The European Rail Traffic Management System (ERTMS) aims to standardize railway signaling systems across Europe by establishing a system of standards for their management and interoperation. ERTMS is divided into several levels to address different needs of railways in Europe:

- **ERTMS Level 0**: No trackside signaling equipment, apart from Eurobalises used for level transitions, and the train is controlled entirely by the driver.
- **ERTMS Level 1**: Trackside signaling equipment communicates with the train to provide speed and distance information. The train is controlled by the driver, and the system enforces a safe speed limit.
- **ERTMS Level 2**: Trackside signaling equipment communicates with an on-board computer that continuously calculates a safe speed limit for the train. The train is controlled by the computer.
- **ERTMS Level 3**: Advanced radio-based train control systems and high level of automation, with no trackside signaling equipment required. The train is controlled entirely by the on-board computer, which communicates with a central control system to ensure safe and efficient train operation.

Several factors determine which level to use, including the presence of another signaling system on the line, the use of GSM-R technology, the maximum speed limit of the line, and any planned capacity upgrades(Abed, 2010).

4.1.2 What is Moving Block?

Moving block refers to a modern railway signaling system that relies on electronic sensors and software to determine the position of each train, facilitating safe and closer train operations with reduced distances between them, which leads to more efficient track capacity. Figure 7 visualizes the idea of the moving block, where the safe separation between trains is determined dynamically based on their actual positions and speeds, resulting in minimal space being wasted. Moreover, the system continuously adjusts the optimal speeds and braking points by taking into account the speed of the preceding train, resulting in a smoother operation of the train's engine and enhanced energy efficiency (Ryan, 2010).

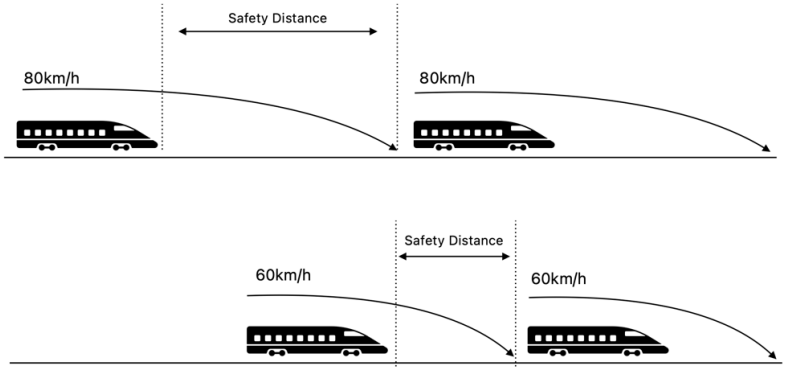

Figure 7: Moving Block – movement authority and safety distance.

A train that is travelling is granted continuous permission to move to the next available section of the track, with the onboard equipment enforcing the latest stop point and maximum speed. This safe distance that the train can travel, determined by factors such as the location of other trains, speed limits, and potential hazards, is known as Movement Authority (MA), which is essential for safe and efficient train operations. The movement authority is granted by a computer-based system called Radio Block Centre (RBC), which uses radio communication to transmit data to and receive data from the trains and trackside equipment (Abed, 2010).

4.1.3 Moving Block Application: Requirements & Constraints

The aim of the use case is to demonstrate a safety critical system running on cloud infrastructure. Therefore, to begin with, the following actors have been identified:

- **Train(s):** The train is the main entity that will be using the train control system. It is responsible for sending its current position and speed to the Moving block application and receiving movement authority in return.
- **Moving Block Application:** The moving block application is a computation system running on the cloud that will grant movement authority to each train. It will receive the current position and speed of each train and issue movement authority in return.
- **Control Center Application:** The control center application is responsible for providing overall management of the railway network. It will maintain track information such as zones and maximum speeds per zone, monitor the network, and require access to the granted movement authority, the current location of each train, and their speed.

Given the actors, following requirements/constraints have been identified:

ID	Requirement
R1	The train control system shall include a computation system running on the cloud, referred to as the moving block application.
R2	The moving block application shall grant movement authority to each train, only if track is free.
R3	Granted movement authority shall be persisted in durable storage.
R4	Each train shall be responsible for sending its current position and speed to the moving block application.
R5	Each train shall receive movement authority in return from the moving block application via HTTP(S).
R6	The control center application shall maintain track information such as zones and maximum speeds per zone.
R7	Track and zone information shall be persisted in durable storage.

R8	The control center application shall require access to the granted movement authority, the current location of each train, and their speed.
R9	Each train shall be equipped with redundant internet connectivity from different telecommunication providers, such as dual SIM from different providers, to communicate with the cloud solution.

Table 5: Use-case requirements

Furthermore, it is assumed that other trackside equipment such as Eurobalises are in place and functional.

4.1.4 Initial setup of Moving Block

The initial system design is provided to become the basis for applying design patterns that are identified. Components did not have any redundancy of any sort, and the system was treated as a single black box. The aim of the process was to start with a rudimentary system to identify all possible design patterns, without making any assumptions about the system under consideration. Therefore, it is assumed that the system is deployed to a single availability zone within a single region, where it connects to the database and the application successfully. Communication takes places over HTTP(S) protocol.

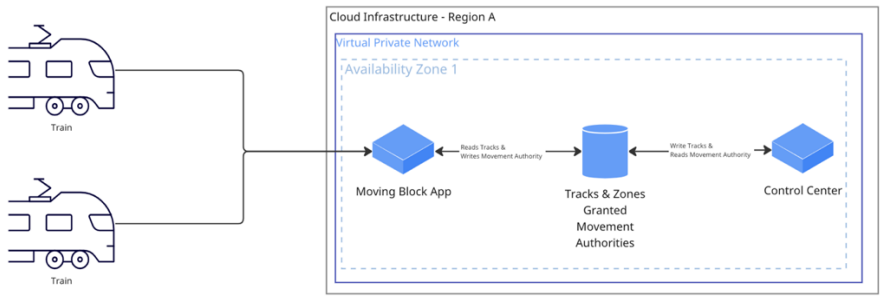

Figure 8: Moving Block: Initial system.

4.1.5 Application Runtime

Cloud providers offer different ways to deploy and run software applications in the cloud. To name a few of them:

A **virtual machine (VM)** is a software-based emulation of a physical computer, capable of executing a full-fledged operating system. Virtual machines (VMs) provide a great degree of versatility and authority, giving you the ability to install and customize any software according to your needs. However, they also demand more administration and upkeep than other alternatives (Tischler, 2021).

Kubernetes is a container orchestration system, that streamlines the automation of application deployment, scaling, and management. Containers offer a streamlined and uniform approach to bundle and roll out software, facilitating the transfer of applications between various environments. Kubernetes offers many different features for administering containerized applications, such as load distribution, automatic scaling, and self-healing (Tischler, 2021).

Serverless functions, commonly referred to as Function-as-a-Service (FaaS), provide the capability to deploy small code snippets that are triggered by events or requests, relieving the burden of managing the underlying infrastructure (Tischler, 2021).

Each of these offers has advantages and disadvantages. In the context of this thesis, application runtime is treated abstractly and is referred to simply as "runtime", since all design patterns can be applied irrespective of the underlying runtime. However, it should also be noted that a particular technology may have an impact on the scalability or maintainability of the entire system.

4.2 Design Patterns

This section presents design patterns in a structured manner, with each pattern including its context, the problem it addresses, its solution, benefits, drawbacks, and an example. A use case is also employed to illustrate the design pattern.

Design patterns listed below were identified during SWIFT sessions. For the complete list of what-if questions, recommended actions, and possible design patterns to address the identified problems, please refer to Appendix 8.2.

4.2.1 Critical Enclave

Context & Problem

Complexity and interdependence can arise in complex systems when different concerns or functionalities are tightly coupled. Without proper separation of concerns, changes or modifications made to one part of the system can have unintended effects on other parts, leading to decreased flexibility, maintainability, and scalability of the system. In the context of safety-critical system, segregating the services based on their criticality level can ensure that any failures in the non-safety-critical services do not affect the safety-critical services.

Solution

This pattern defines an approach to make a clear boundary of safety-critical system and non-safety-critical system. Essentially, it is based on a separation-of-concern principle that aims to separate a complex system into distinct and independent parts, each of which addresses a specific concern or functionality. Hence, it requires a clear separation of compute, storage, and networking concerns for safety-critical system and non-safety-critical system. Since most systems are interdependent and not isolated, if a safety-critical application needs to interact with a non-safety-critical system, this can be accomplished by establishing a clearly identified interface and implementing strict rules to prevent any unauthorized access that could render the safety-critical service unavailable.

Although there are various ways to achieve this, one simple solution is to use the existing tools provided by cloud providers. Most cloud providers offer ways to organize and manage resources, such as AWS Accounts, GCP Projects, and Azure Subscriptions, which can be easily utilized to create the separation of concerns.

Benefits

The benefits of this design pattern are:

- **Flexibility:** By separating concerns and functionalities in a complex system, it becomes easier to make changes or modifications to specific parts without affecting other components. This flexibility allows for more agile development and adaptation to evolving requirements.
- **Maintainability:** Clear separation of concerns enables easier maintenance and troubleshooting. Developers can focus on specific areas without the need to understand

the entire system, making it simpler to identify and fix issues. This results in improved system stability and reduces the time required for maintenance tasks.

- **Testability:** With clear boundaries and well-defined interfaces between components, it becomes easier to create test cases that cover specific functionalities or interactions. Testing efforts can be focused on critical areas or components, ensuring comprehensive coverage, and reducing the risk of overlooking potential issues.
- **Isolated Failures:** The failure of one subsystem or component does not affect other components, increasing system resilience.

Issues and Considerations

Following points should be considered for this design pattern:

- **Integration Challenges:** In situations where safety-critical and non-safety-critical systems need to interact, establishing and managing a clearly defined interface can pose integration challenges. Ensuring proper communication and data exchange between the segregated components requires careful coordination and adherence to strict rules.

Example

In the use case of moving block application, anything that is essential to the system has to be separated. Control center application needs to be able to alter the track information, such as zones and speed limit as well as access to current train positions. However, this access can still be provided either by clearly defined interfaces. Nevertheless, the separation of components should be the first step to accomplish, enabling easier maintenance and testing of the system. In the following Figure 9, this has been visualized.

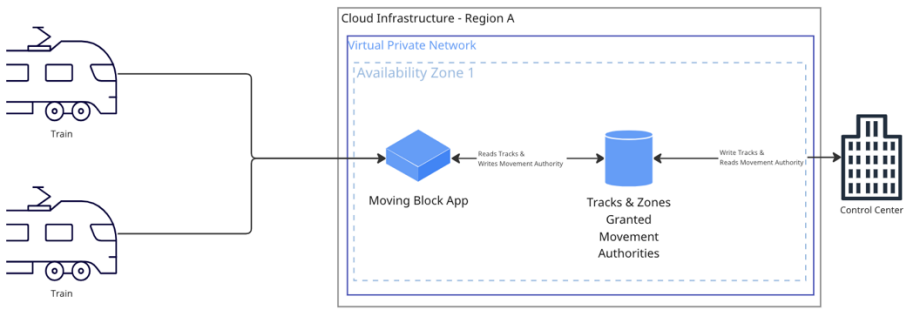

Figure 9: Critical enclave pattern applied to the use case.

4.2.2 Data Segmentation

Context & Problem

Traditionally, most systems had a single database system where all data that was required for the operation of the system was persisted in a single storage, irrelevant of their read/write profile or individual attributes. This approach was consistent, as most systems were developed in monolithic style of software development, one big application would connect to one big database. Furthermore, allowing access to the data as well as permission to update the data has been problematic, as circumventing process execution in the scope of a single application is rather trivial.

Solution

With the advancement of cloud technologies, DevOps practices, as well as microservice architecture, the behavior that was described has shifted towards abstracting the database behind a service interface. This has resulted in the use of databases with different storage technologies (e.g., NoSQL, relational, graph) for different purposes, as well as clearly defined boundaries. Hence, this pattern addresses the issue by dividing a larger dataset into smaller, more specific subsets based on certain criteria or attributes. The goal of data segmentation is to enable better performance for read/write profiles, as well as to isolate subsets for each use case that have common characteristics or behaviors (Newman, 2021; Tischler, 2021).

Benefits

The benefits of this design pattern are:

- **Performance:** Read/write profiles can be optimized for each subset, resulting in improved database performance.
- **Scalability:** By isolating subsets with common characteristics or behaviors, data segmentation enables better scalability for each use case. This means that as the system grows, the database can be easily scaled up or down to meet changing needs.

- **Maintainability:** With clearly defined boundaries for each data subset, maintenance and updates become easier, as changes made to one subset are less likely to impact other subsets.
- **Security:** Access to each subset can be controlled and monitored more easily, reducing the risk of unauthorized access or data breaches(Newman, 2021).

Issues and Considerations

Following points should be considered for this design pattern:

- **Complexity:** It requires additional development effort, either to create code for accessing multiple databases in a single application or to create and manage service interfaces to access the data.
- **Data Consistency:** With distributed data, the same information is often copied in different data stores for different reasons. Data propagation and consistency can become a challenge to manage.
- **Management Overhead:** Having an increased number of databases requires management overhead, such as backup and recovery, monitoring, and maintenance.

Example

In the use case of the moving block application, the dataset can be divided into two subsets: the track and zones, the first dataset and the granted movement authority, the second dataset. The control center application creates and updates the track information, so the moving block application only needs read permission to access the first data set. On the other hand, the moving block application generates granted movement authorities, so it requires read/write permissions to the second dataset. The control center application only needs read permission to access the second dataset. Additionally, the scalability requirements for both subsets are different. The track information is not updated as often as the movement authority and train position, so second dataset is a read-heavy dataset. In the following Figure 10, this has been visualized.

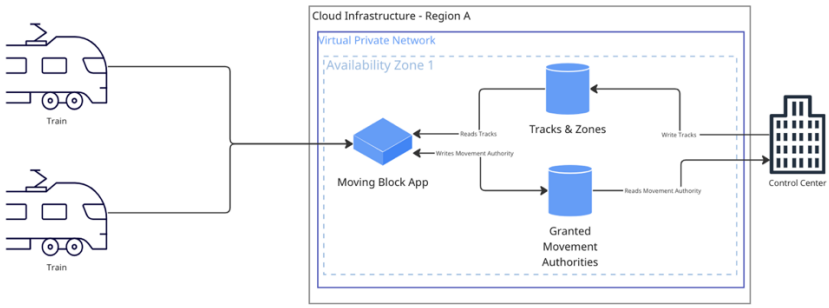

Figure 10: Data segmentation pattern applied to the use case.

4.2.3 Publish–Subscribe Pattern

Context & Problem

Traditional tightly coupled client-server paradigm depends on the fact that the server must be running for the client to successfully complete its pending operation, and the server cannot process operations if the client is not running. Furthermore, the client-server paradigm can have an unintended consequence where clients may overload the server, resulting in failures or causing delays in more critical processes.

Solution

The publish-subscribe (pub/sub) pattern is often used in event-driven architectures, where a component publishes a message to a channel without any knowledge of who will receive it, or whether anyone will receive it. Other components can subscribe to the channel to receive those messages and take actions that are usually independent of the original message sender. The fundamental design principle revolves around components communicating with one another without any awareness of each other's presence or existence. This loosely coupled architecture allows for independent operation of the components (Malaska & Seidman, 2018).

Benefits

The benefits of this design pattern are:

- **Loosely Coupled Architecture:** Components communicating with one another without any awareness of each other's presence or existence.

- **Scalability:** New subscribers can be added without affecting the publisher or other subscribers, allowing for easy expansion.
- **Flexibility:** The system allows for the seamless addition or removal of components without causing any impact on the remainder of the system, thereby enabling enhanced adaptability.
- **Isolated Failures:** The failure of one component does not affect other components, increasing system resilience (Malaska & Seidman, 2018).

Issues and Considerations

Following points should be considered for this design pattern:

- **Complexity:** Introducing a messaging system adds another component to the system, which can increase its complexity.
- **Latency:** As pub/sub communication is asynchronous, there may be increased latency between when a message is published and when it is processed by the subscriber.
- **Testing and Debugging:** Testing and debugging a pub/sub system may be difficult due to its complex interactions between multiple components.

Example

In the use case of the moving block application, the control center application requires the current position and speed of trains on the track. Although this information is available in the database, direct access by the control center application through regular requests to the database can cause processing time to be used by the control-center, rather than the actual safety-critical part of the application which is moving block application. In the context of the system, moving block application has a higher priority to use and exclusively manage the database to avoid any possible failure propagating.

The information needed by the control center application can still be delivered via a pub-sub mechanism. Once the moving block application approves the Movement Authority (MA) and persists it in the database, it can send the result to the pub/sub system. This results in low decoupling of the system as well as separation of safety-critical and non-safety-critical systems.

Furthermore, as every cloud provider provides a messaging middleware, the maintenance of such a component is relatively low. In the following Figure 11, this has been visualized.

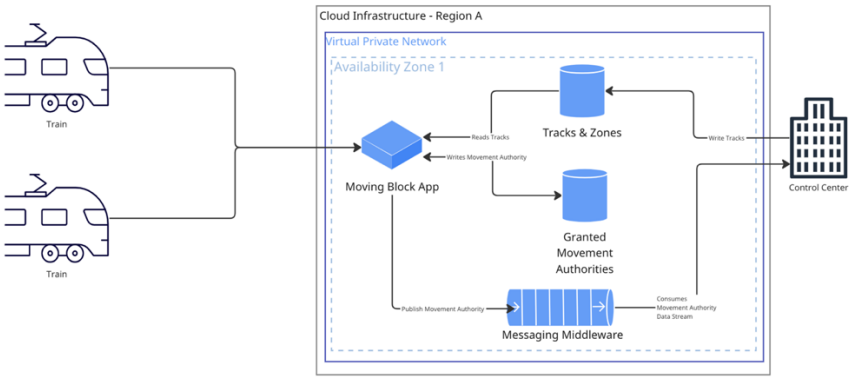

Figure 11: Publish-subscribe pattern applied to the use case.

It is worth to note that, similar decoupling effects can also be achieved using other methods, such as providing a read-only replica of the database to the control center application. Current data replication technologies are mature enough to handle such synchronization with minimal latency.

4.2.4 Stateless Computation

Context & Problem

Distributing a workload among different computational units requires consideration of several factors such as the complexity of the task, the capabilities of the individual units, and the need for synchronization and communication between them. When a workload maintains internal state, it becomes challenging to distribute the workload across multiple servers because the state must be synchronized across all instances.

Solution

Stateless Computation design patterns dictate that the workload should be designed in a way that it does not maintain any internal state. To implement this approach, the workload must be decomposed into smaller units of work, and each unit should be designed to be self-contained and independent of the others. It is important to explicitly provide inputs for each unit and return the output explicitly. If the output is required for further computation, it can be stored in an

external database. Furthermore, the design should ensure that any necessary state information is passed as an input to the unit instead of being stored within the unit. This approach allows each server to access the required state when needed, without the need for synchronization or communication with other servers (Newman, 2021).

Adopting this pattern simplifies the distribution of the workload across multiple servers, and scaling the workload up or down by adding or removing servers becomes effortless, thereby enhancing overall performance and efficiency.

Benefits

The benefit of this design pattern is:

- **Scalability**: As the computation unit does not have any internal state, it can scale up and down in a very simple manner.

Issues and Considerations

Following point should be considered for this design pattern:

- **Communication Overhead:** Stateless computation may result in increased network traffic due to the need to pass state information between services and databases.

Example

In the use case of moving block application, the application can turn into a stateless one by relying on external storage. Below is the diagram that outlines this idea: the application can be initialized with no data, and as soon as the request comes in, the required data can be fetched from the database. Once the operation is completed, the data is persisted, and no information is kept on the service. In any case, each request can behave as if there was no data, and once the request is completed, the result can be persisted. In the following Figure 12, this has been visualized.

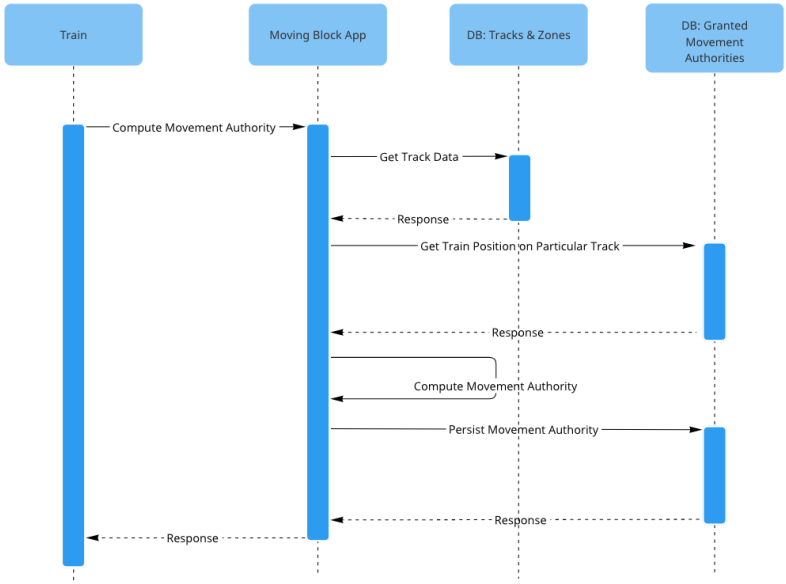

Figure 12: Stateless computation pattern applied to the use case.

4.2.5 Multi-Availability Zone

Context & Problem

Traditionally, applications were developed, deployed, and run in a single on-premises data center. Redundancy was achieved by running the application on multiple virtual machines within the same data center. If the data center were to experience an outage or disruption, the application or service could become unavailable, potentially causing significant harm to the business. Multiple data centers were utilized for segmentation of work, backup, and for situations such as disaster recovery.

Solution

An Availability Zone (AZ) is a concept in cloud computing that refers to a physically separate data center within a geographical region that is designed to provide high availability and fault-tolerance. Each Availability Zone typically consists of one or more data centers that are geographically dispersed to reduce the risk of a single point of failure. The Multi-Availability Zone pattern involves deploying an application or service across multiple AZs within a single region of a cloud provider's infrastructure. Each AZ is designed to be isolated from failures in other AZs, which means that if one AZ were to experience an outage or disruption, applications

and services running in other AZs would remain unaffected. In summary, redundancy is achieved by replicating an application and data across multiple data centers to protect against failures (Tischler, 2021).

Benefits

The benefit of this design pattern is:

- **Availability**: Multi-availability zone ensures high application availability with redundant infrastructure in multiple locations. If one zone fails, traffic redirects to another automatically.

Issues and Considerations

Following point should be considered for this design pattern:

- **Data Consistency:** While the replication of the stateless computation resources is rather easy, applications must be designed with data consistency in mind, as having multiple availability zones means data is replicated across multiple data centers, which can introduce data consistency issues because of latency.

Example

In the use case of moving block application, redundancy and improved availability would be achieved by deploying the application in different availability zones. Generally, each region would have at least three different availability zones, hence the advice would be to deploy the application at least once to each availability zone in a region. As applications would be independent of each other, the need for a load balancer arises, which is used as an entry point to distribute the workload across the applications. Furthermore, the same redundancy and improved availability can be applied to the databases as well. In this case, databases will use an internal replication mechanism to replicate the data across availability zones. In this example databases and messaging middleware are considered regional resources. The assumption here is that these databases and messaging middleware are cloud provider solutions that only need to be configured, unlike the computational resources, which require explicit design. In the following Figure 13, this has been visualized.

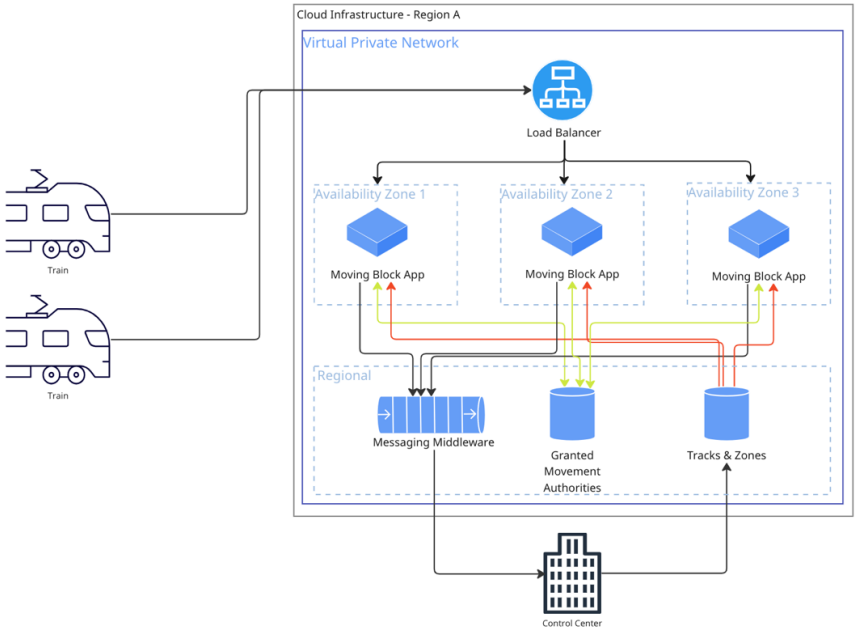

Figure 13: Multi-availability zone pattern applied to the use case.

4.2.6 Auto-Scaling

Context & Problem

The traditional approach to scaling applications has been to manually provision and manage infrastructure resources, which can be time-consuming and error prone. In this approach, engineers would have to manually provision new instances and resources as demand increased, and de-provision them as demand decreased. This approach is not only inefficient but can also lead to underutilization or over-provisioning of resources, which can impact performance and cost.

Solution

Auto-scaling is a feature in cloud computing that automatically adjusts the allocation of resources within a cloud environment, adapting to changes in demand. By dynamically scaling resources up or down, it enhances availability, minimizes costs, ensuring efficient resource utilization. Scaling resources can be done via different metrics that are collected from the group of resources such as CPU, memory, or network traffic. Auto-scaling can be used to help ensure that new instances of applications are launched with the right configuration and settings, while load balancers can help distribute traffic and requests evenly across different availability zones

and instances. The auto-scaling feature is used in many different application runtimes, and most cloud providers offer auto-scaling for virtual machines, functions, or containerized runtime. Furthermore, in some cases, such as using technologies like Kubernetes, autoscaling is achieved by scaling both the underlying virtual machines and the workload that runs on Kubernetes. Therefore, based on the runtime choice, autoscaling is applied at multiple levels.

When it comes to scaling an application, there are two main approaches: horizontal scaling and vertical scaling. Horizontal scaling involves adding more instances of the application to share the workload across multiple machines, while vertical scaling involves adding more resources (such as adding more CPU, memory, or storage) to a single instance of the application. Based on best practices, it is generally recommended to rely more heavily on horizontal scaling. This is because horizontal scaling offers better fault tolerance, is easier to scale in response to changing demand, and is typically more cost-effective compared to vertical scaling (Tischler, 2021).

Benefits

The benefits of this design pattern are:

- **Scalability**: In case of a demand for computational resources, the system can scale up and handle the demand, keeping the availability of the system high.
- **Reduced Costs:** It can help reduce costs by only using the necessary number of resources, scaling down when demand decreases, as well as the manual effort that engineers would have otherwise spent on maintaining the system.

Issues and Considerations

Following point should be considered for this design pattern:

- **Delay**: Auto-scaling may experience some delay depending on the underlying runtime and number of resources that are required. For example, launching a new virtual machine can take several minutes, but containerized applications have an advantage as they tend to auto-scale more quickly. This difference in speed is due to the underlying technology. However, even containerized applications require virtual machines, as Kubernetes needs virtual machines as nodes to run the containerized applications.

Although scaling a specific workload on Kubernetes is faster, if there are insufficient resources, Kubernetes will also require additional virtual machines to handle auto-scaling of containerized applications. Essentially facing similar constraints as virtual machines.

Example

In the use case of moving block application, assuming application is already using stateless computation design pattern, memory usage of the application would be relatively stable, however for the calculation of movement authority it would require more CPU power than memory. In this case, autoscaling can be based on CPU usage, for instance if average CPU utilization is above 80% application can be scaled out. The new instance of the application, soon as starts accepting request from the load balancer, the average CPU utilization would go down. This process is visualized in Figure 14 and 15. In Figure 14 concurrent load of the system is shown and how the load is impacting average CPU utilization, horizontal line representing the time. As the concurrent load increases (from *t1* to *t3*), the average CPU utilization also increases (from 66% to 83%). Once the average CPU utilization of all resources increases the above the threshold of 80%, new instance of the application is created and made available. At *t4*, even though the concurrent load remains the same as *t3* (1000 request per second), the average CPU utilization drops to 62.5%, the reason for this is that new instance of the application is now present and sharing the workload. This resource increase is visualized in Figure 15 at *t4*. At *t6*, when the concurrent load increases to 1400 request per second, the average CPU utilization increases to 87.5%, triggering another scale-up to 5 instances. The same procedure in reverse applies causing system to scale-down.

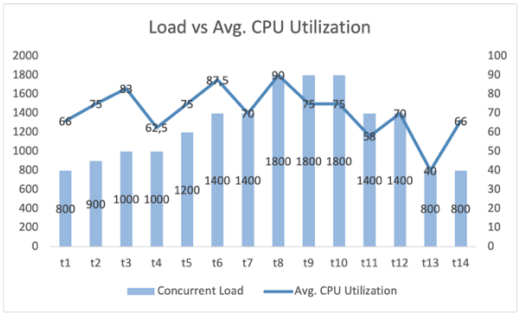

Figure 14: Load vs avg. CPU utilization

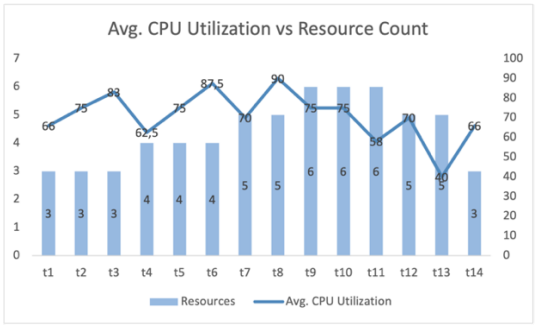

Figure 15: Resource count vs avg. CPU utilization

4.2.7 N-Version Programming and Deployment

Context & Problem

It is generally accepted that developing 100% fault-free software is impractical. Traditional software development methods rely on testing and debugging to identify and remove faults, but this approach has limitations, as testing all possible scenarios and configurations is difficult. Furthermore, even if the computation itself is correct, it is essential to consider the potential impact of bit-flip errors occurring in memory. These errors can introduce corruption into the response, leading to inaccurate or unreliable results. Bit-flip errors can be caused by various factors, including cosmic radiation, electrical interference, or inherent flaws in memory modules, potentially compromising the integrity and accuracy of the system's output.

Solution

N-version programming addresses this problem by creating multiple independent versions of the same software or system, with each version developed by a different team or group of developers. N-version programming reduces the likelihood that all versions will contain the same faults by using different algorithms and programming languages. The results produced by each version are compared, and if they differ, the most commonly occurring result is assumed to be correct (Mukwevho & Celik, 2021).

To implement this pattern, an orchestration of distributing the same input to different versions of components, gathering their results, and making a final decision based on a voting mechanism is required. If the application requires additional information from the database, the orchestrator should retrieve it before processing begins and distribute the input to different computational units. This ensures that computations do not access the database at different times and retrieve inconsistent data. If the computation result needs to be persisted, the orchestrator should connect to the database and handle the persistence operation. This approach also has the added benefit that, if one of the requests fails, the orchestrator or voter can handle the failure gracefully and make another request. In the following Figure 16, this has been visualized.

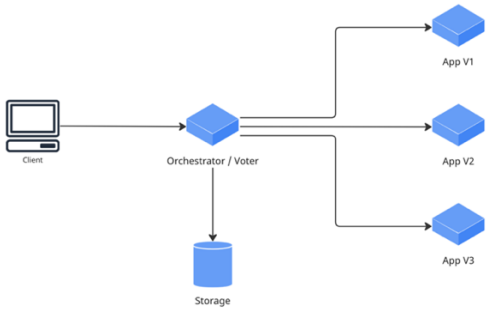

Figure 16: N-Version programming with voter/orchestrator

Benefits

The benefits of this design pattern are:

- **Fault Tolerance:** Because each version of the software is designed independently, they are less likely to share the same flaws or vulnerabilities. This can make the software more resilient to unexpected errors or attacks.

Issues and Considerations

Following points should be considered for this design pattern:

- **Development Cost:** Creating multiple independent versions of the same software component can be time-consuming and costly.
- **Complexity:** Orchestrator introduces a new level of complexity and can become the bottleneck of the system. Furthermore, each computational unit has to be redundant independently of other versions, further increasing the complexity.

Example

In the use case of moving block application, the infrastructure view would be as follows: different versions of the app would be deployed on different availability zones, and the orchestrator/voter would also be replicated. Once a request arrives, the orchestrator would mirror the request to different versions of the computational unit, collect the results, perform the voting procedure, and persist the result in the database, as well as respond to the train. Essentially stripping all storage access functionality from moving block application and only

making it responsible for the computation of the movement authority. In the following Figure 17, this has been visualized, for the sake of brevity, a single request flow has been depicted. Request is routed from the load balancer to an orchestrator, which in the example, it lands on the Availability Zone 3. Orchestrator makes the request to the closest resources, which are in the same zone, if one of the requests fails, orchestrator can make a request to the resources in the other zones. Ensuring that response is collected successfully.

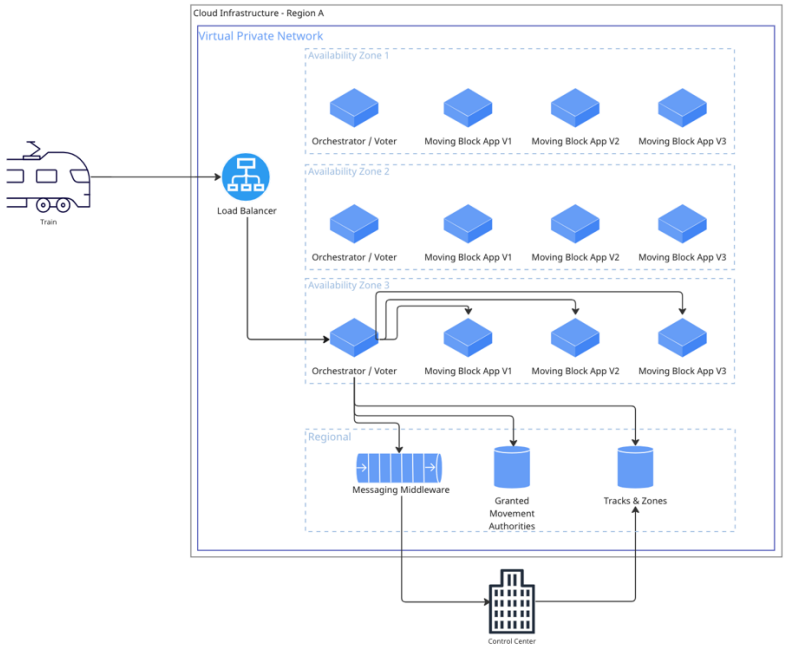

Figure 17: N-version programming pattern applied to the use case.

4.2.8 API Gateway

Context & Problem

Most of the latency in communication between a client and a cloud-based application occurs during the establishment of the connection and the round-trip of requests between the two. In the event that a request made by the client fails, the client is required to re-initiate the request, which naturally nearly doubles the overall response time for a particular request. Similarly, requests that take too long to complete may cause delays and impact on the overall performance of the system. In both of these cases, the client is impacted by the potential delay and has the responsibility to handle errors as well as timeout requests.

Solution

One potential solution to these challenges is to add time-out and retry logic to each service. However, this approach can result in duplicated code and make the system more complex. Alternatively, this logic can be centralized, simplifying the codebase, and making it easier to manage and maintain. As well as making the overall system much more resilient as the failures will not propagate to the client. Hence, the API Gateway pattern aggregates common operations and adds an additional layer of error handling (Newman, 2021).

The API Gateway, which serves as an entry point for all requests coming from external clients and routes internally to responsible services to handle the request. While the connection from the client is open to API Gateway, there is another connection opened to the component which does the actual computation. If the component fails to response in time, or the request fails, API Gateway can place another request with the aim of getting a timely response. From client perspective however, there is a single connection and system response timely without failure (Newman, 2021).

Benefits

The benefits of this design pattern are:

- **Fault Tolerance:** Timeouts and other system failures are encapsulated by the retry mechanism that has been implemented, which effectively manages control flow of requests, ensuring that the system remains stable and performs efficiently.
- **Shared Common Responsibilities:** As the API Gateway is the central entry point, other common responsibilities such as authentication/authorization can be handled by this component, reducing the amount of code required in the remaining parts of the system.
- **Rate-limiting:** The API Gateway can establish a fine-grained control over the rate at which requests are allowed, ensuring controlled access to the underlying resources. This preventive measure helps safeguard against potential abuse or misuse of the system, as it restricts the number of requests an entity can make within a given time frame by enforcing sensible limits (Newman, 2021).

Issues and Considerations

Following point should be considered for this design pattern:

- **Single Point of Failure:** Due to its central role in the system, it becomes a single point of failure, hence it has to be designed very carefully.

Example

In the use case of moving block application, the API Gateway would be placed after the central load balancer, handling all request routing, timeouts, and retries of the requests. Other responsibilities such as authentication and authorization can also be offloaded, protecting the rest of the system. In the following Figure 17, this has been visualized, for the sake of brevity, a single request flow has been depicted. The API Gateway first makes a request to a resource that is placed in Availability Zone 2. If the request fails or takes too long to respond, the API Gateway can make another request to a different resource, as depicted in the figure, where the request is being handled by the resources in Availability Zone 3. All cloud providers offer similar API Gateway services. Hence, this is the reason why the API Gateway component has been depicted outside of the Availability Zone. However, it is also very common that existing solutions do not fulfill all requirements, leading to the need for choosing a custom-developed API Gateway.

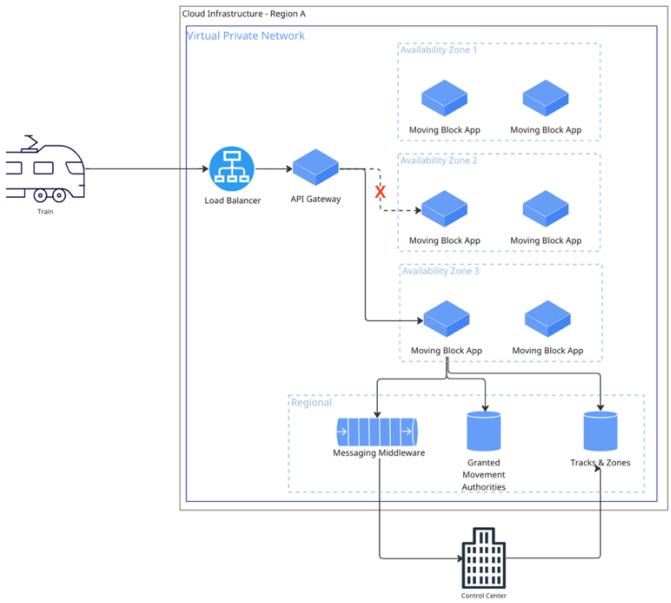

Figure 18: API gateway pattern applied to the use case.

4.2.9 Multi-Region Deployment

Context & Problem

Cloud providers typically have measures in place to mitigate the impact of regional outages, such as redundant systems and backup data centers located in different geographic regions. However, a variety of factors, such as natural disasters, power outages, network failures, hardware failures, or other unforeseen events can still cause regional outages. Furthermore, human errors, such as misconfiguration (e.g., accidentally disabling a critical network switch or firewall, causing all traffic to be blocked), can also cause a regional outage, although there may not be any physical damage to the data center or its infrastructure. In section 2.4.3, more information about regional outages has been provided.

Solution

The multi-region deployment pattern suggests deploying the workload in two or more separate regions to protect against service outages and disruptions in a particular region. Multi-region architectures distribute the application across multiple geographic regions, allowing users to access the application from the region closest to them. This improves performance and reduces latency. Additionally, in the event of a regional outage or disruption, the application can continue to function in other regions.

While the multi-region deployment pattern may appear similar to the multi-availability zone pattern, it actually provides an additional layer of redundancy that complements the latter. Therefore, both patterns are commonly used in conjunction to enhance the availability and resilience of cloud applications. It is worth noting, however, that the multi-region deployment pattern can be more challenging to implement compared to the multi-availability zone pattern (*AWS Multi-Region Fundamentals - AWS Whitepaper*, 2022; *Azure Regions Decision Guide - Cloud Adoption Framework*, 2023).

Benefits

The benefits of this design pattern are:

- **Availability:** In the event of an outage or disaster in one region, services in other regions can still continue to function.
- **Performance:** It can improve latency for users who are geographically distributed. By allowing users to access services and resources from the region closest to them, the design can result in lower latency and faster response times.

Issues and Considerations

Following points should be considered for this design pattern:

- **Complexity:** A multi-region design can be more complex to implement and manage compared to a single-region design. Ensuring that resources are properly distributed and synchronized across multiple regions can require additional infrastructure and operational resources.
- **Cost:** A multi-region design can be more expensive to implement and maintain compared to a single-region design. For instance, additional infrastructure, data transfer costs, and other expenses are some of the factors that can increase the cost.
- **Latency:** Although the overall latency of the systems will not be high, there are some applications where the impact of latency becomes most apparent. This is particularly true for databases that rely on the ability to efficiently share their commit log among different nodes or replicas. In such scenarios, any delay in propagating the commit log updates across regions can result in synchronization issues, data inconsistencies, or even potential data loss.

Example

In the use case of moving block application, the application would be deployed in at least two separate regions, with a global load balancer distributing traffic between them. Instead of traditional round-robin load balancing, which distributes traffic to each server in turn, global load balancers typically route traffic based on regional proximity or other values, with the goal of sending clients to the same region for a consistent user experience. While data replication remains a significant concern for multi-region deployment, there are various methods and techniques available to address data synchronization. If a regional outage occurs, the multi-region deployment pattern allows for traffic to be automatically redirected to the remaining

available region, ensuring the continuity of service. However, it is crucial to consider data consistency during the design phase, as well as how the system will handle regional outages and recovery. The ability to seamlessly handle regional outages and maintain uninterrupted service is a critical aspect of multi-region deployment, especially for safety-critical applications that cannot afford any downtime. In the following Figure 19, this has been visualized. Computational elements are replicated across region; however, all computational resources do connect to Region A for database access. Although same resources do exist in Region B, for better performance most systems tend to have a single region as primary data storage and secondary region as backup. In case of regional outage, backup storages would be activated to continue operation.

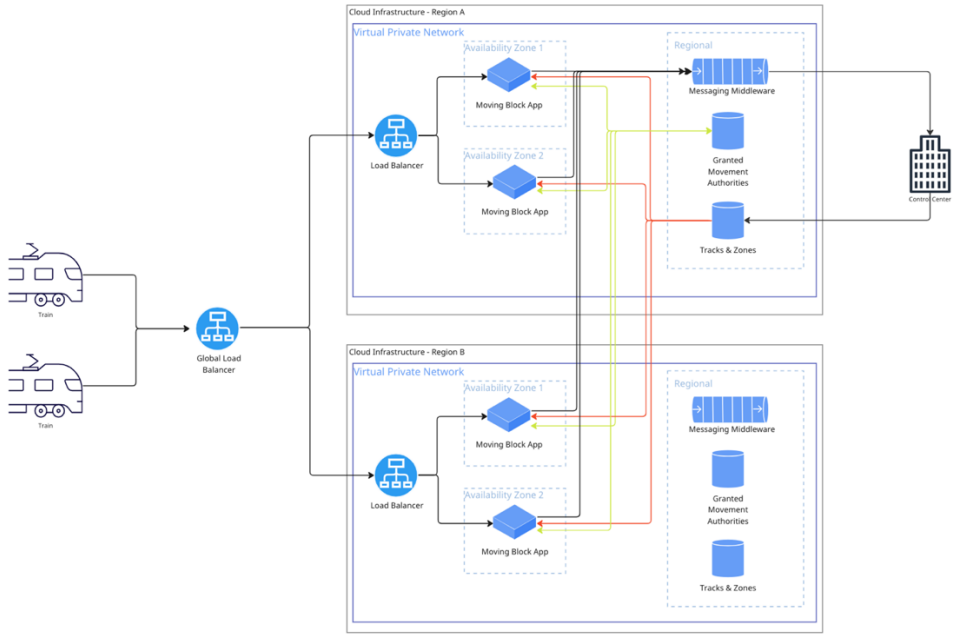

Figure 19: Multi-region deployment pattern applied to the use case.

4.2.10 Geo-Replication

Context & Problem

Running a multi-region, highly available system poses a significant challenge: ensuring continuous data availability and accessibility despite network outages, hardware failures, and other disruptions. In a multi-region deployment, if one region becomes unavailable, achieving redundancy in computational power is relatively straightforward, as processes can operate

independently. However, ensuring redundant data becomes much more difficult due to physical limitations.

Solution

The geo-replication pattern involves the deliberate choice of an appropriate database technology that offers robust geo-replication capability to ensure the reliability and resilience. With geo-replication, the selected database technology ensures that data is replicated and synchronized across geographically dispersed sites or regions in near real-time. This safeguards against potential disruptions, such as network outages, hardware failures, or natural disasters, by ensuring that the system remains operational even in the face of localized incidents. The aim is to preserve data integrity and minimize the risk of data loss.

Cloud providers offer a diverse range of database technologies, each with its own distinct methods of implementing geo-replication. These technologies provide organizations with flexibility in choosing the most suitable option based on their specific requirements and preferences. Some database technologies employ synchronous replication, where data is immediately and consistently replicated across multiple regions, ensuring strong data consistency but potentially introducing additional latency. Other technologies utilize asynchronous replication, where data is replicated with a slight delay, offering higher scalability and potentially lower latency for read access in different region but with a trade-off of eventual consistency.

Benefits

The benefits of this design pattern are:

- **Ease of Access:** Implementing and managing geo-replicated databases demands a considerable level of expertise, which may not be readily available. However, cloud providers possess this expertise and offer geo-replication services typically accompanied by higher service level agreements (SLAs).
- **Data Availability:** Data is replicated across multiple geographic locations, enhancing availability, and reducing the risk of data loss, in case of regional outages, or natural events.

- **Scalability:** By distributing data across multiple regions, it can potentially accommodate increased workload demands.

Issues and Considerations

Following points should be considered for this design pattern:

- **Performance**: While geo-replication can improve performance in terms of latency reduction, it can also introduce performance variability. Factors such as network congestion, data transfer speeds between regions, and synchronization delays can impact the overall performance of the replicated system. It is crucial to design the system in a way to tolerate physical limits of data transmission (Malaska & Seidman, 2018).
- **Vendor Lock-in:** Since the database used in geo-replication is usually proprietary and not open-source, it becomes highly improbable to run the same software on a different cloud provider. This poses a challenge for organizations that opt for a multi-cloud strategy, as different cloud providers may employ different technologies, making it difficult to achieve geo-replication across multiple providers.
- **Cost:** Utilizing geo-replication across multiple regions can incur additional costs for data transfer, storage, and infrastructure in each region.

Example

In the use case of a moving block application, the application setup would be very similar to a multi-region deployment, where applications are deployed in at least two separate regions. A global load balancer would distribute traffic between these regions. Applications in each region will communicate with the data storage and services located in that particular region. However, data synchronization will occur between the data storages, replicating data back and forth between the regions. In Figure 20 below, this has been visualized, with replication depicted by orange lines. In the diagram, the control center application connects only to a single region. In the event of a regional outage, the control center must change its connection to an available region, but data will still be replicated.

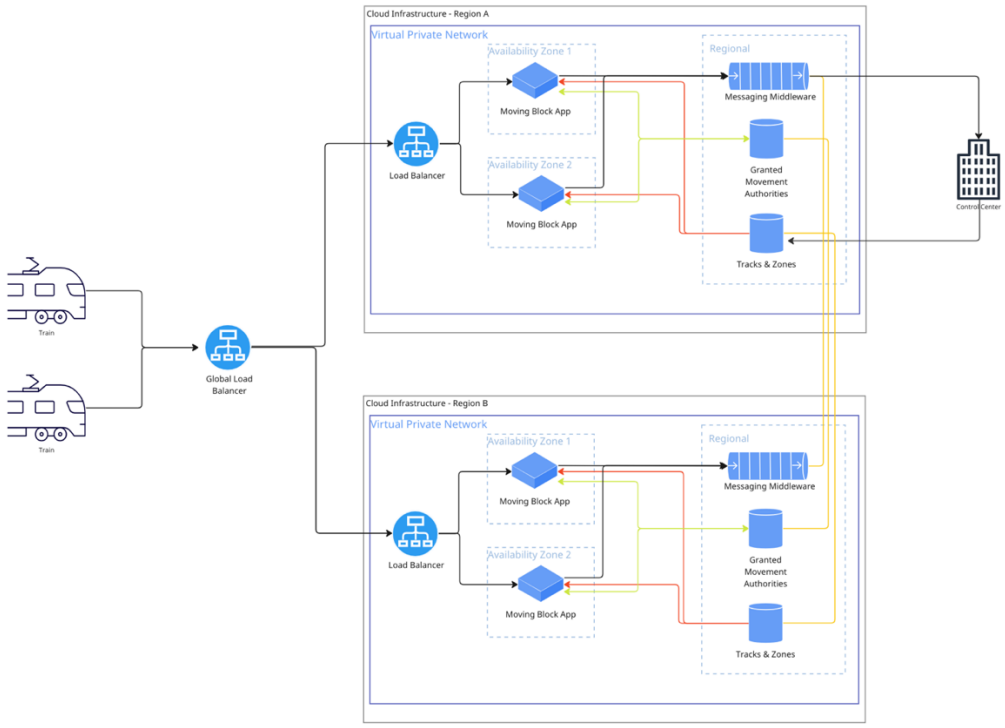

Figure 20: Geo-replication pattern applied to the use case.

4.2.11 Elastic Workload Segmentation

Context & Problem

When implementing a multi-region deployment in an active-active manner with geo-replication, one of the primary concerns is the difficulty of achieving strong consistency across all regions. Strong consistency ensures that all replicas of data in different regions are kept synchronized and up to date. However, achieving this level of consistency poses very hard challenges (Malaska & Seidman, 2018).

When data is transmitted over long distances between geographically separate regions, certain inherent limitations arise, resulting in latency within the system. This latency emerges due to the necessity of replicating and synchronizing data across multiple regions, which consumes time and impacts overall system performance and responsiveness. Consequently, critical operations like reading and writing data, as well as real-time interactions with the application, are affected. The presence of network delays, varying network conditions, and potential conflicts during data replication further exacerbate the issue, leading to inconsistencies across different regions (Malaska & Seidman, 2018).

Solution

In the elastic workload segmentation design pattern, the workload of the system is distributed among different regions in an elastic manner. This means that requests from the same set of clients always end up in the same region as long as the region is healthy. If one of the regions becomes unavailable, all the requests are routed to the remaining healthy region. Data is still replicated between the regions as the basis of eventual consistency. However, since the same set of clients always ends up in the same region, there is data locality. In the event of a regional outage, there is a risk of a small amount of data loss due to the replication method being eventual consistency.

Cloud providers offer global load balancers that provide various traffic routing methods. These methods include latency or proximity-based routing, prioritizing services handling requests, weighted distribution, session affinity, and routing based on HTTP(S) parameters such as host, path, and headers. If none of these methods suffice, each region can have a unique DNS name for resolution, although this requires the client to switch destinations on demand. Cloud provider solutions alleviate the need for custom code development and simplify client-side implementation since they connect to a single destination.

Benefits

The benefits of this design pattern are:

- **Availability:** With the workload distributed across multiple regions, the system can achieve higher availability. In the event of a regional outage, the workload is automatically shifted to the remaining healthy region, minimizing service disruptions, and ensuring continuous availability.
- **Resilience:** The use of a multi-region deployment in an active-active manner enhances the system's resilience. If one region becomes unavailable, all requests are automatically routed to the remaining healthy region, ensuring continuity of service.
- **Performance**: Prioritizing data locality significantly improves user latency by storing data closer to users or their respective regions. This reduces data retrieval and processing time, resulting in faster response times and improved system performance. However, achieving higher availability may require adopting an eventual consistency model.

Temporary data inconsistencies across regions are resolved over time. Despite this trade-off, prioritizing data locality greatly enhances user experience by minimizing latency and enabling quicker access to relevant data (Malaska & Seidman, 2018).

Issues and Considerations

Following points should be considered for this design pattern:

- **Consistency:** The replication method used in this design pattern is based on eventual consistency, which means that there might be a delay in synchronizing data between regions. This introduces the risk of data inconsistencies and a small amount of data loss during regional outages (Malaska & Seidman, 2018).
- **Data Loss Risk:** Due to the eventual consistency approach, there is a potential risk of data loss during regional outages. If a region becomes unavailable before data replication occurs, any updates or changes made in that region might not be synchronized with the remaining healthy region, resulting in data loss.
- **Complexity:** Managing a multi-region deployment and ensuring consistent replication of data adds complexity to the system architecture. It requires careful configuration, monitoring, and synchronization mechanisms to maintain data integrity and minimize the risk of inconsistencies.

Example

In the use case of a moving block application, the application setup would extend the multi-region deployment model with geo-replication. A global load balancer would distribute traffic among these regions. However, the distribution of traffic would be based on arbitrary criteria, such as the track the train is on and the train's location on the track. The addressed issue is that if two trains are following each other on the same track, they should land on the same region to avoid any data consistency issues. This is visualized in Figure 21 below, where Client Set A requests land on Region A (depicted with gray lines), while Client Set B requests land on Region B (depicted with blue lines).

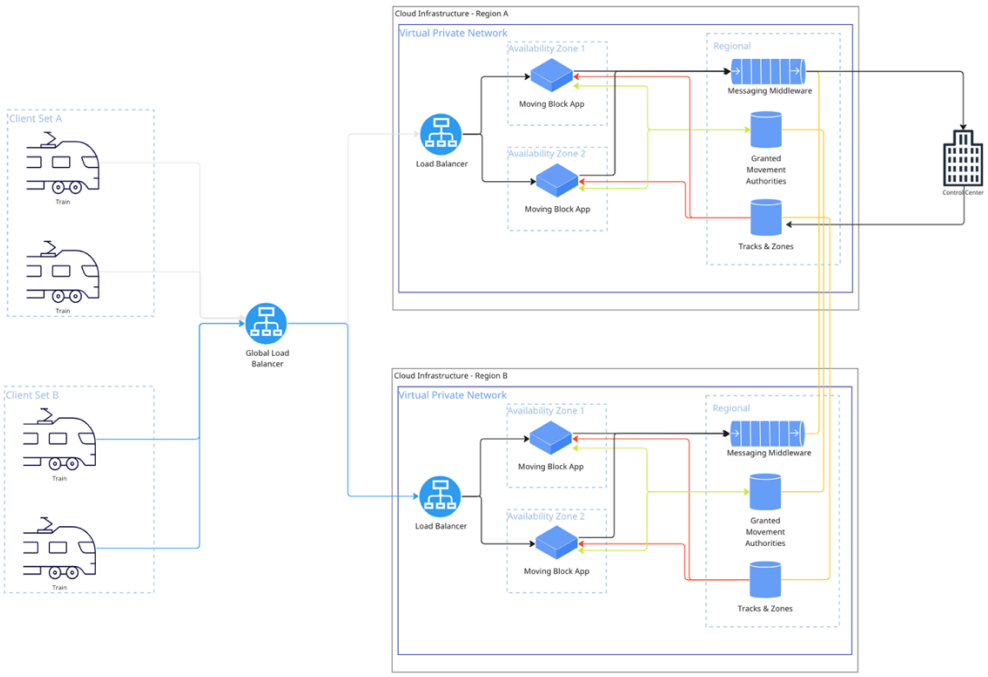

Figure 21: Elastic workload segmentation pattern applied to the use case.

Naturally, the assumption here is that a global load balancer can implement such logic. However, given the use case and available properties of a global load balancer, not all cloud providers may have such a feature. In such a case, some of the responsibilities can be offloaded to the client, such as resolving the available regions before the journey starts and communicating directly with that region. This idea is illustrated in Figure 22, where the client (Train A) makes an initial request to a Config Server to receive all available region information, as well as determining the preferred region. Afterwards, it starts operating normally by connecting to Region A and requesting computation of movement authority. If there is an outage or for some reason Region A is not reachable, the client tries to communicate with Region B. This approach is much more flexible because it also allows dynamic changing which region a client can connect to. For the use case, it is particularly useful as once a train crosses certain zones on a track, it may make much more sense to communicate with a different region. The entire system image is depicted in Figure 23.

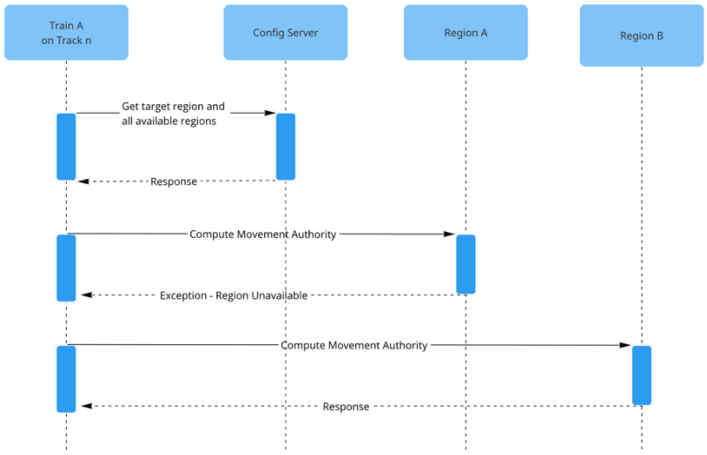

Figure 22: Client switching region in elastic workload segmentation.

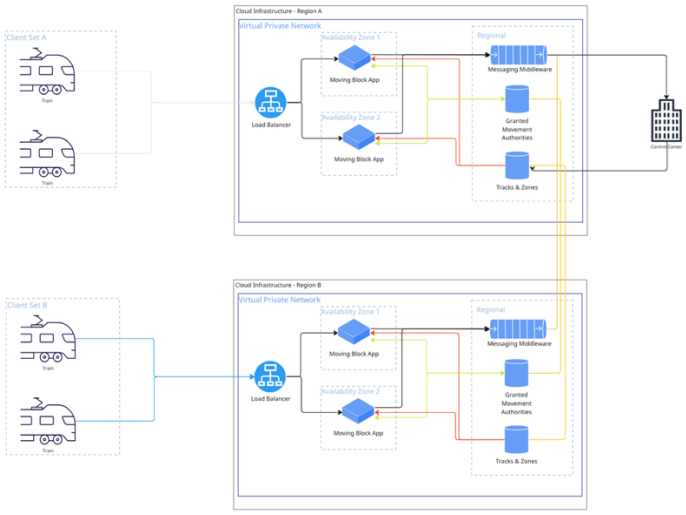

Figure 23: Elastic workload segmentation pattern without global load balancer applied to the use case.

4.2.12 Multi Cloud Deployment

Context & Problem

While deploying workloads across multiple regions within a single cloud provider can help to address some of these challenges, it may not be sufficient to meet all the requirements of the organization. In addition, a single cloud provider outage can cause a significant disruption to

business operations, potentially resulting in financial losses and damage to the organization's reputation. In section 2.4.3, the issue of regional outage has been described in detail.

Solution

To overcome these challenges, organizations can embrace a multi-cloud strategy, which entails distributing workloads across multiple cloud providers. This approach offers several advantages, including enhanced availability, geographic redundancy, and flexibility. However, the most significant benefit of utilizing multiple cloud providers is the mitigation of downtime risks associated with a single provider outage. In this pattern, workloads are deployed to multiple cloud providers, and requests are intelligently redirected to different providers based on predetermined criteria. By implementing this approach, organizations can ensure uninterrupted service delivery, optimize performance, and maintain a robust infrastructure that can adapt to changing needs and circumstances.

There are two distinct approaches available for implementing this solution. The first approach involves selecting a global load balancer to distribute the workload across multiple cloud providers. However, this approach can be quite challenging to achieve due to the requirement of synchronizing data between the different providers. Alternatively, the second approach entails partitioning the workload among different cloud providers. In the event of a failure or outage with one provider, all traffic is then redirected to the available provider. It is important to note that this approach assumes the presence of effective data synchronization mechanisms between the various cloud providers (Mulder, 2020).

Benefits

The benefits of this design pattern are:

- **Increased Availability:** Deploying workloads across multiple cloud providers improves the overall availability of services. If one cloud provider experiences an outage, the organization can rely on the other provider(s) to ensure business continuity and minimize disruptions.
- **Flexibility and Vendor Lock-In Avoidance:** Deploying workloads across multiple cloud providers offers flexibility and avoids vendor lock-in (Mulder, 2020).

Issues and Considerations

Following points should be considered for this design pattern:

- **Complexity:** Managing multiple cloud providers introduces complexity in terms of infrastructure, deployments, and operations. It requires expertise in working with different provider-specific tools, APIs, and configurations.
- **Data Portability:** Ensuring data portability, particularly replication across multiple cloud providers, can be highly challenging. Data solutions with built-in geo-replication capabilities are typically not compatible across different cloud providers. Therefore, organizations must implement custom data synchronization and replication solutions to address this issue effectively.
- **Application Portability:** Different providers may have variations in services, APIs, and data storage models, requiring organizations to design and develop applications with portability in mind.
- **Interoperability and Integration:** Integrating and ensuring interoperability between different cloud providers can be complex. Organizations may face challenges in achieving seamless communication and data exchange between workloads deployed across multiple clouds. Compatibility issues, different networking architectures, and security configurations must be addressed to ensure smooth operations and data flow.
- **Network Complexity and Costs:** Deploying workloads across multiple cloud providers requires robust and reliable network connectivity between the providers. Organizations must invest in networking solutions, such as VPNs or direct connections, to establish secure and high-bandwidth connections. This can increase network complexity and costs associated with data transfer (Mulder, 2020).

Example

In the use case of a moving block application, application setup and structure would be very similar to elastic workload segmentation. The only difference would be regions would be from different cloud providers and geo-replication has to be custom develop instead of using cloud providers solutions.

4.2.13 Redundant DNS

Context & Problem

DNS providers, just like any other service, can encounter technical issues, network outages, hardware failures, software glitches, or become the target of cyber-attacks. Failures of DNS providers can have multiple consequences, including DNS resolution failures, service disruptions, DNS caching issues resulting in incorrect resolutions, and delays in updating DNS records. These issues can directly affect the accessibility of destination sites, online services, and the timely propagation of changes.

Solution

To further mitigate the impact of potential DNS provider failures, organizations can implement a redundancy strategy by leveraging multiple DNS providers. This approach involves creating multiple domain names from different DNS providers to ensure failover capabilities. When resolving host names, the response from the host resolution can be configured in two ways.

Firstly, it can point to a single address, which could be a global load balancer. This load balancer would then distribute the incoming traffic to the appropriate regions or instances based on predetermined criteria. This approach ensures that even if one DNS provider fails, the load balancer can continue to direct traffic to the available resources. Alternatively, the response can consist of multiple addresses, with each entry pointing to a different region or instance.

In any case, clients are designed to be aware of all the domain names associated with the various DNS providers. If there is a problem with host name resolution due to the failure of one DNS provider, clients can seamlessly switch over to the next available domain name and continue accessing the desired resources.

By implementing this redundancy strategy with multiple DNS providers and configuring the host resolution responses accordingly, organizations can significantly enhance the resilience and availability of their systems. Clients can continue to access the services even in the event of a DNS provider failure, ensuring uninterrupted connectivity.

Benefits

The benefits of this design pattern are:

- **Reliability & Availability:** Redundancy in DNS infrastructure ensures that DNS resolution remains available even if one provider or server becomes unavailable. This helps maintain the accessibility of destination sites and online services, preventing potential downtime and ensuring uninterrupted access for users.

Issues and Considerations

Following points should be considered for this design pattern:

- **Complexity on Client Side:** Clients have to be aware of multiple DNS entries and switch if one becomes unavailable, the error handling part especially DNS caching issues and have to be considered.
- **Configuration Complexity:** Managing multiple DNS providers or deploying secondary DNS servers adds complexity to the DNS infrastructure configuration. It requires careful setup, synchronization, and monitoring to ensure consistency and proper failover mechanisms.

Example

In the use case of a moving block application, each client will have a set of domain names to which clients can connect. Figure 24 illustrates overall system, with clients are using two domain names, namely *api.railway1.com* and *api.railway2.com*, which are registered and managed by different providers. The infrastructure view has been simplified. In this example, clients can use different domain names to connect to same system, hence the global load balancer handles the distribution of the traffic. Alternatively, each domain name can be associated with a single region, as depicted in the Figure 25. The most important part is the error handling that occurs on the client side. When a client encounters a problem with resolving a DNS or receives an error from the target, the client can switch to the secondary domain name and retry the operation.

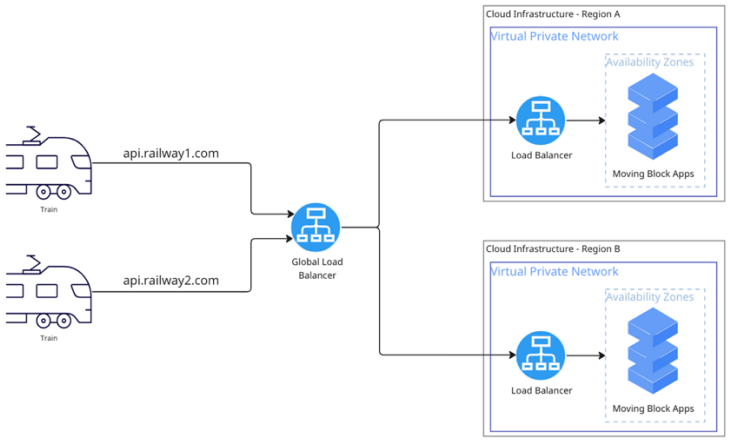

Figure 24: Redundant DNS pattern with global load balancer applied to the use case.

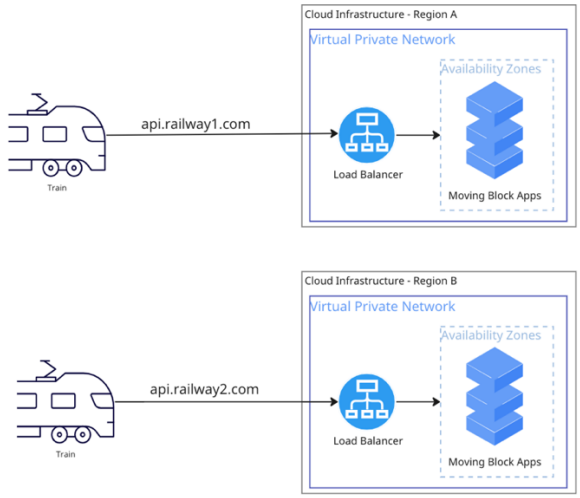

Figure 25: Redundant DNS pattern each entry assigned to a region applied to the use case.

4.3 Essential Practices for Safety-Critical Systems

While the aforementioned design patterns provide a solid foundation, there is a need for additional practices to enhance system safety and performance. These supplementary measures encompass crucial aspects such as observability, monitoring, logging, alerting, and more. The combined implementation of these practices ensures a comprehensive approach to availability and reliability of safety-critical systems. Therefore, this section will provide a brief summary

of these practices to emphasize the importance of utilizing the above design patterns in conjunction with the following practices.

- **Service Health Check:** This refers to the process of periodically verifying the health and availability of a service. It involves performing checks to ensure that the service is functioning properly and can handle incoming requests. Each service can expose an endpoint to provide its health. This is particularly important when services depend on other components, such as databases, as there may be a delay between the service starting to run and the time it takes to connect to the database. Therefore, the health check provides essential information that the service is running and operational (Bass et al., 2015).
- **Monitoring:** Monitoring involves the continuous observation and measurement of various system metrics, including the current load of the system, available capacity, performance, and other indicators, to be able to detect anomalies. The overall goal is to ensure that the system operates within desired thresholds (Bass et al., 2015).
- **Logging:** Logging involves capturing and recording important events, actions, and data within a system, as well as errors and failures. It serves as a valuable source of information for troubleshooting, debugging, and auditing purposes. The primary purpose of logs is to identify issues and analyze system behavior (Bass et al., 2015).
- **Alerting:** Alerting is the practice of sending notifications or alerts when certain predefined conditions or thresholds are met. It enables timely response to critical events, anomalies, or errors, allowing engineers to take appropriate actions to mitigate the issues that are either occurring or about to occur (Bass et al., 2015).
- **Continuous Integration and Continuous Delivery (CI/CD):** CI/CD is a software development practice that focuses on automating the process of code integration, building, and deploying software to different environments, including production. The aim is to be able to release software in a reliable manner, reducing manual effort and creating a faster feedback loop. CI/CD includes many automated tests in place for the release of the software, which increases the quality of the whole system. Furthermore, CI/CD also helps to roll back the faulty changes that have been deployed quickly, making the recovery process much easier and faster (Bass et al., 2015).

- **Infrastructure as Code:** Infrastructure as code involves defining and managing infrastructure resources using automation tools and code, with the aim of producing repeatable provisioning of infrastructure, making it easier to manage, version control, and reproduce environments (Bass et al., 2015).
- **Back-up Procedures:** This refers to implementing appropriate backup strategies and procedures to ensure data integrity and availability, by backing up important data and systems. It is also important to be able store backups securely and have practices in place for testing the restoration process (Tischler, 2021)..
- **Planned Failovers:** This refers to deliberately but in a controlled manner taking part of the system down, forcing failover procedure. During a controlled failover test, a specific component, subsystem, or portion of the system (e.g., a region) is intentionally taken offline. This action triggers the failover process, where either the workload and associated resources are transferred to a backup or secondary system or incoming traffic is redirected to redundant system. The purpose of this test is to evaluate whether the failover mechanisms and restore procedures function as intended and can successfully handle the transition of operations to mitigate the impact of unplanned downtime and reduce service interruptions (Tischler, 2021)..
- **Collaboration and Quality Control:** This refers to implementing a structured review process for system-impacting changes, such as software modifications, software deployment, and infrastructure as code updates. The aim is to promote collaboration and ensure the maintenance of high code quality. By carefully reviewing and approving changes before deployment, the integrity and reliability of software development projects are upheld (Tischler, 2021).

4.4 Conclusion

In this chapter, a safety-critical use case has been introduced, utilizing SWIFT, a prospective hazards analysis method, to identify a set of issues and potential solutions. These solutions have then been transformed into design patterns that can be reused in other systems. Each design pattern has been documented with its context, the problem it addresses, the proposed solution, as well as its benefits, drawbacks, and an example. The use case has been employed to demonstrate an example of the design pattern. In some cases, the design pattern has become an integral part of the new system design, evolving with each example.

5 Evaluation of Identified Design Patterns for System Unavailability

The evaluation section of this thesis aims to assess the effectiveness of the identified design patterns for addressing system unavailability in the moving block use case that was introduced in section 4.1. For this purpose, several of the identified design patterns in section 4.2 will be used together to create a system design for the moving block use case. Finally, an analysis of system unavailability using Fault Tree Analysis (FTA) will be conducted based on the generated system design and the cloud computing failure modes described in 2.4.1.

5.1 System Design: Moving Block Use Case

In this section, some of the identified design patterns are combined to create a system design for the moving block use case. The selection of the design patterns has been based on a versatile choice, covering most of the issues that were identified in section 3.5 Data Collection and Analysis: Generating What-If Questions, while still aiming to keep the system design simple enough for analysis. All the identified issues and recommended actions can be found in appendix 8.2. The following nine design patterns that were described in section 4.2 have been selected for the system design: 4.2.1 Critical Enclave, 4.2.2 Data Segmentation, 4.2.3 Publish–Subscribe Pattern, 4.2.4 Stateless Computation, 4.2.5 Multi-Availability Zone, 4.2.6 N-Version Programming and Deployment, 4.2.9 Multi-Region Deployment, 4.2.10 Geo-Replication, 4.2.11 Elastic Workload Segmentation.

The system design, as illustrated in Figure 26, depicts the architecture and components of the proposed solution. The system is deployed in two regions, with each component being redundant and deployed at least once across three different availability zones. The system relies on turn-key solutions provided by the cloud provider, including load balancers, databases, and messaging middleware. During the initial operation, each train connects to a random region and receives information about the track and zone details, as well as the geographical computation region it should connect to during the journey. The connection of each train to the designated geographical computation region during the journey is visually represented by the orange line in Figure 26. Once the train receives the information, it can start requesting movement authority to initiate the journey. During their journey, trains connect to a specific region. This method

has been chosen to minimize potential data replication issues that may arise. If one of the regions fails, it is the responsibility of the train to connect to the second available region. As data is replicated between regions, the overall system can continue to operate even if one of the regions becomes unavailable.

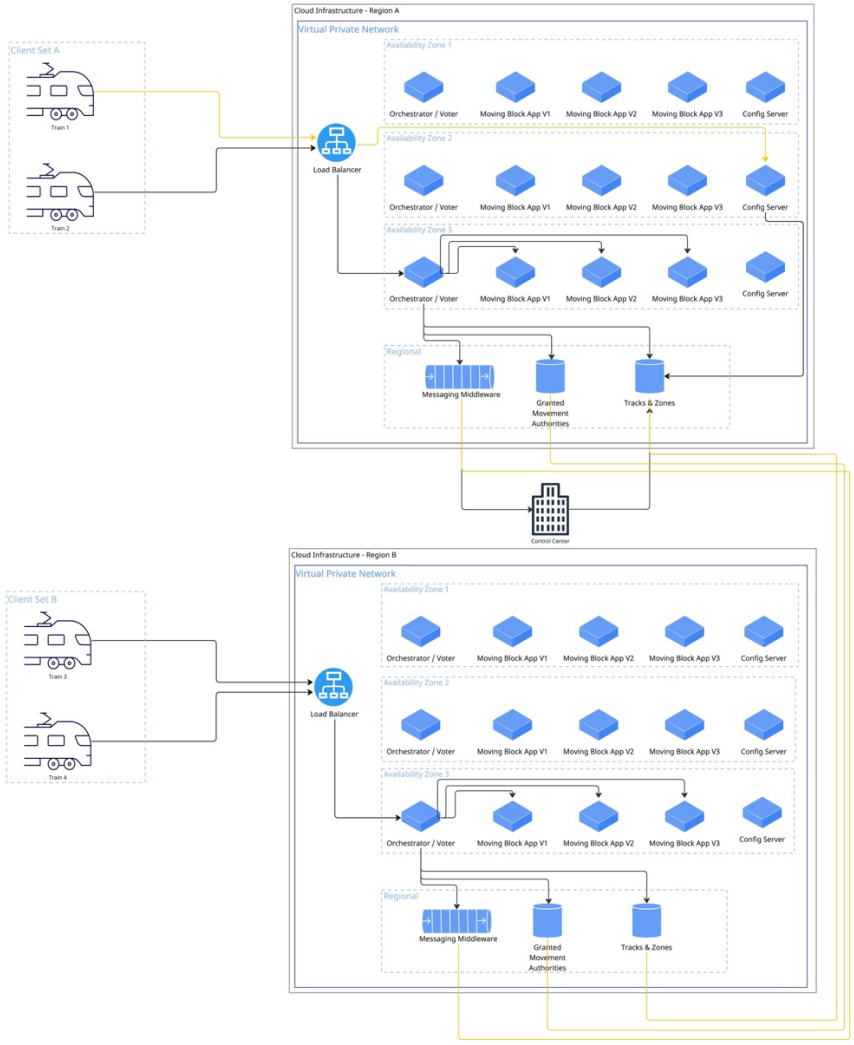

Figure 26: Composed design patterns applies to the use case.

5.2 Analyzing System Unavailability with Fault Tree Analysis

Fault Tree Analysis (FTA) will be employed in section 5.1 to thoroughly examine the proposed system design and assess its ability to mitigate system unavailability. However, considering the extensive range of potential failures in cloud computing, encompassing hardware (such as CPU,

memory, disk, and network switch), software, cloud management, environmental factors, and more, it is impractical to account for every individual failure scenario.

To address this challenge, one approach is to abstract failures and treat groups of failures as single events. This allows us to focus on the overall impact of failure events rather than getting lost in the specifics of each individual failure. For example, if an application fails to complete its task due to a hardware issue, whether the failure originated from a disk failure, CPU malfunction, or any other hardware-related problem becomes irrelevant. What matters is the higher-level failure that affects the application's performance and availability, as that is what cloud consumers are primarily concerned with.

Furthermore, cloud service providers often offer Service-Level Agreements (SLAs), which are contractual agreements defining the expected level of service between providers and customers. SLAs typically outline various aspects of the service, including availability, uptime, response time, resolution time, and support hours. Leveraging SLAs can provide a means to estimate failure probabilities. For instance, if an SLA guarantees a specific uptime percentage for a virtual machine (VM), it is reasonable to assign a corresponding probability to the event of VM failure based on the reliability information specified in the SLA. Similar estimation can be applied to databases, load balancers, and other components by considering their respective SLAs.

By abstracting events and utilizing SLAs, it becomes feasible to conduct a comprehensive Fault Tree Analysis (FTA). This approach allows us to capture the broader failure scenarios while incorporating the probability estimates derived from SLAs. By doing so, we can gain a deeper understanding of the critical factors contributing to system unavailability and evaluate the effectiveness of the proposed system design in mitigating these risks.

The Fault Tree Analysis (FTA) is visualized in Figure 27, showing a fault tree diagram of the moving block system. The diagram represents the logical relationships and dependencies between events causing system unavailability. The top event in the fault tree diagram is ***system becomes unavailable*** with an estimated probability of 0.016573. This corresponds to a system SLA of 99.983427%. Considering a daily downtime of 14 seconds, the estimated monthly downtime for the system is 7 minutes, and 12 seconds. This calculation provides insights into

system reliability. Further sections will describe the technology selection, potential errors that covered, deriving failure probabilities from SLAs.

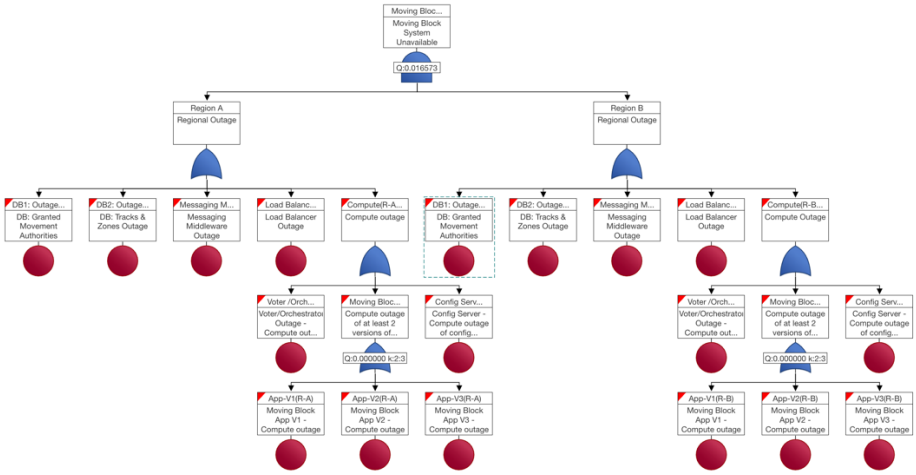

Figure 27: Fault tree analysis of moving block system

The Appendix 8.3 contains the FTA analysis in the form of a table. The cut set analysis performed in Appendix 8.4 indicates that the total probability exceeds 0.01, while in Appendix 8.5, the total probability surpasses 0.001.

5.2.1 Technology Selection and SLAs

For the purpose of obtaining realistic estimations and SLAs, Microsoft Azure has been selected as the cloud provider. Cosmos DB has been chosen for data storage, which offers a 99.999% SLA and provides geo-replication capabilities. Service Bus has been selected as the messaging middleware with a 99.9% SLA. Additionally, Load Balancer and Virtual Machines have been chosen with SLAs of 99.99% each (*Service Level Agreement for Microsoft Online Services*, 2023).

The SLAs mentioned in the above have specific requirements. For example, the 99.99% SLA for virtual machines is guaranteed when at least two virtual machines across two availability zones in the same region have been started. Therefore, in the Fault Tree Analysis (FTA) diagram, each service is assigned a separate failure probability instead of assigning a single probability to all virtual machines. Same requirements also apply to the Cosmos DB, meaning

for SLA of 99.999%, database has to be span in multiple Azure regions with multiple writable locations (*Service Level Agreement for Microsoft Online Services*, 2023).

Furthermore, each SLA has an extensive definition that specifies the cases in which downtime is calculated. For example, in the case of virtual machines, the SLA encompasses not only hardware failures but also virtual machine connectivity. Similarly, for databases, the SLA considers not only service uptime but also failed requests, which are counted towards the SLA calculation.

Given the SLAs failure probability for the FTA is calculated, as follow:

$$Failure\ Probability = 100 - Service\ SLA$$

Finally, the probability is converted to daily possible downtime as follow.

$$Daily\ Downtime = \frac{Seconds\ in\ a\ day}{100} * Failure\ Probability$$

5.2.2 Limitation of Relying on SLAs

It is important to note that SLAs represent contractual agreements and may not provide the exact failure probabilities. They can serve as a starting point for estimation, but additional data and analysis may be required to derive accurate probabilities. Historical data, vendor documentation, expert judgment, and other sources of information can be used to refine the probabilities within the fault tree.

5.3 Conclusion

In this chapter, effectiveness of design patterns in addressing system unavailability in the moving block use case. Fault Tree Analysis (FTA) provided insights into the system's vulnerability to failures and guided the proposed system design. By abstracting failures and using SLAs, the evaluation prioritized overall impact and estimated failure probabilities. The FTA diagram highlighted the logical relationships between failure events, with system unavailability as the main focus.

6 Conclusion

In this final chapter, the study will conclude by summarizing the main research findings in relation to the research aim and questions. Additionally, it will discuss the significance and contribution of these findings. The chapter will also acknowledge the limitations of the study and present potential avenues for future research.

6.1 Findings

This thesis aimed to create a collection of architectural design patterns that can be used by various safety-critical systems that run on public cloud infrastructure. Furthermore, based on research aim and objectives following research questions were identified:

- **RQ1:** Can all safety-critical systems be deployed to the public cloud?
- **RQ2:** What are the existing fault-tolerance methods in the cloud?
- **RQ3:** How to identify relevant design-pattern?
- **RQ4:** Can design pattern be used in various safety-critical systems?

To achieve the research aim of developing architectural design patterns for safety-critical systems on public cloud infrastructure, a systematic approach was followed. Multiple safety-critical use cases were considered, and the railway signaling system, specifically the moving block computation, was chosen as the most suitable use case. The selection of this use case took into account critical factors such as timing constraints and system resilience. Safety-critical systems that can tolerate a delay in response and have the capability to re-request computations were prioritized. In the case of railway systems, trains are equipped with additional safety features that provide redundancy and mitigate risks. Therefore, even in the event of a total failure of the cloud system, the potential harm is theoretically minimized. In section 2.2.3, the aspect of latency was discussed in detail.

Number of fault tolerance methods as well as cloud failure modes has been identified and reviewed, which were further utilized to identify design patterns, these can be found in section 2.4.1 and 2.5.

In order to identify relevant design patterns, a prospective hazards analysis method called Structured What-If Technique (SWIFT) was utilized. This analysis helped identify various issues and recommended actions to address them. These recommended actions were then mapped onto design patterns that could be applied across different projects, ensuring their wide applicability.

Finally, this thesis argues that these design patterns can be used in various safety-critical projects. Each pattern presents a clear problem statement and provides guidelines for implementing a solution, along with the associated benefits and drawbacks.

6.2 Contribution

The resulting collection of design patterns offers a valuable resource for architects and engineers working on safety-critical systems in the cloud. By leveraging these patterns, they can enhance system reliability, mitigate risks, and improve overall performance.

The design patterns offer practical solutions to address specific problems and provide a framework for the design and implementation of robust and secure systems. Additionally, the documentation of each design pattern with detailed information, including context, benefits, drawbacks, and practical examples, adds further value by facilitating understanding and adoption.

The developed collection of design patterns can be readily applied in both research and practical settings. In research, these patterns serve as a foundation for further exploration and analysis of safety-critical systems in the cloud. Researchers can investigate the effectiveness of these patterns, refine them, or develop new patterns based on the established framework. In practice, architects and engineers can directly apply these design patterns in their projects, adapting them to specific requirements and leveraging the documented guidance to ensure proper implementation.

Furthermore, the generated what-if questions and guidewords, also provide a valuable starting point for project teams, facilitating the identification of potential risks and guiding the decision-making process. Overall, the application of these design patterns contributes to the advancement of safety and reliability in cloud-based safety-critical systems.

6.3 Limitation of This Study

Furthermore, it is important to acknowledge that this study has focused on a specific scope and may not encompass all aspects related to the design and implementation of safety-critical systems on public cloud infrastructure. Confidentiality, integrity, and security are crucial considerations in any system, including those deployed using cloud technologies. However, due to the specific focus of this study, these aspects have not been extensively addressed.

Moreover, the limitations of this study extend to the availability and accessibility of use cases that were used to derive the design patterns. The selection of suitable use cases plays a significant role in the generalizability and applicability of the patterns. While efforts were made to consider a range of use cases, it is important to acknowledge that the patterns may be more applicable to certain contexts and may require adaptation for different scenarios.

Additionally, the use of deductive reasoning in deriving the design patterns introduces a limitation in terms of their generality. The patterns developed in this study are based on logical deductions and may not cover all possible variations and edge cases. The patterns should be considered as a starting point and may require further refinement and customization based on specific system requirements.

Overall, while this study provides valuable insights and guidelines for designing safety-critical systems on public cloud infrastructure, it is essential to recognize its limitations and consider additional factors and considerations specific to individual projects and environments. Continued research and exploration are encouraged to expand and refine the design patterns, addressing the diverse needs and challenges of safety-critical applications in various contexts.

6.4 Further Research Opportunities

Further research is warranted to address the limitations and extend the findings of this study. One important direction for future research is to investigate the integration of confidentiality, integrity, and security aspects into the design patterns for safety-critical systems on public cloud infrastructure. This would ensure a comprehensive approach that takes into account the complete range of system requirements and considerations.

Additionally, there is a need to expand the availability and accessibility of use cases for deriving design patterns. Future research should focus on gathering a diverse set of use cases from different domains and industries, considering a wide range of safety-critical applications. This would enhance the generalizability and applicability of the design patterns and provide more insights into their effectiveness in various contexts.

It is also important to conduct empirical validation and refinement of the design patterns in real-world scenarios. This would involve implementing the patterns in actual safety-critical systems deployed on public cloud infrastructure and evaluating their effectiveness in improving system reliability, availability, and fault tolerance. Such empirical studies would provide valuable insights into the practical application and performance of the design patterns.

In conclusion, continued research and exploration are essential to overcome the limitations identified in this study. By addressing the gaps in the current findings and considering additional factors and considerations specific to individual projects and environments, the design patterns for safety-critical systems on public cloud infrastructure can be expanded and refined. This would ultimately contribute to the development of more robust and adaptable solutions for meeting the diverse needs and challenges of safety-critical applications in various contexts.

6.5 Closing Summary

In conclusion, this study has developed architectural design patterns for safety-critical systems on public cloud infrastructure, focusing on availability and reliability. The collection of design patterns offers practical solutions for architects and engineers. However, limitations include the exclusion of confidentiality, integrity, and security aspects, the need for more diverse use cases,

and the reliance on deductive reasoning. Future research should address these limitations and further refine the design patterns to meet specific project requirements. Continued exploration is crucial for enhancing the design patterns and advancing safety in cloud-based safety-critical systems.

7 References

Abed, S. K. (2010). European Rail Traffic Management System—An overview. *2010 1st International Conference on Energy, Power and Control (EPC-IQ)*, 173–180.

Abelein, U., Lochner, H., Hahn, D., & Straube, S. (2012). Complexity, quality and robustness—The challenges of tomorrow's automotive electronics. *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 870–871. https://doi.org/10.1109/DATE.2012.6176573

Adler, R. (2019). *Dependability Engineering* (2nd ed.). University of Kaiserslautern.

Agarwal, H., & Sharma, A. (2015). A comprehensive survey of Fault Tolerance techniques in Cloud Computing. *2015 International Conference on Computing and Network Communications (CoCoNet)*, 408–413. https://doi.org/10.1109/CoCoNet.2015.7411218

Amin, Z., Singh, H., & Sethi, N. (2015). Review on fault tolerance techniques in cloud computing. *International Journal of Computer Applications*, *116*(18), 11–17.

Ataallah, S. M. A., Nassar, S. M., & Hemayed, E. E. (2015). Fault tolerance in cloud computing—Survey. *2015 11th International Computer Engineering Conference (ICENCO)*, 241–245. https://doi.org/10.1109/ICENCO.2015.7416355

Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, *1*(1), 11–33. https://doi.org/10.1109/TDSC.2004.2

AWS Multi-Region Fundamentals—AWS Whitepaper. (2022).

AWS Well-Architected Framework. (2023, March 28). https://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html

Azure regions decision guide—Cloud Adoption Framework. (2023, June 8). https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/migrate/azure-best-practices/multiple-regions

Bar-Yam, Y. (2002). General features of complex systems. *Encyclopedia of Life Support Systems (EOLSS), UNESCO, EOLSS Publishers, Oxford, UK*, *1*.

Bass, L., Weber, I., & Zhu, L. (2015). *DevOps: A software architect's perspective*. Addison-Wesley Professional.

Birdsong, C., Schuster, P., Carlin, J., Kawano, D., Thompson, W., & Kempenaar, J. (2006). *Test methods and results for sensors in a pre-crash detection system*. SAE Technical Paper.

Card, A., Ward, J., & Clarkson, P. (2012). Beyond FMEA: The structured what-if technique (SWIFT). *Journal of Healthcare Risk Management : The Journal of the American Society for Healthcare Risk Management*, *31*, 23–29. https://doi.org/10.1002/jhrm.20101

Cook, R. I. (1998). How complex systems fail. *Cognitive Technologies Laboratory, University of Chicago. Chicago IL*, 64–118.

Danielsson, J., Tsog, N., & Kunnappilly, A. (2018). A Systematic Mapping Study on Real-Time Cloud Services. *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 245–251.

Eurostat-Cloud computing. (2023). Eurostat. https://ec.europa.eu/eurostat/statistics-explained/SEPDF/cache/37043.pdf

Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Patterns, D. (1995). Elements of Reusable Object-Oriented Software. *Design Patterns*.

Gilbert, S., & Lynch, N. (2012). Perspectives on the CAP Theorem. *Computer*, *45*(2), 30–36. https://doi.org/10.1109/MC.2011.389

Global Infrastructure Regions & AZs. (2023, March 29). Amazon Web Services, Inc. https://aws.amazon.com/about-aws/global-infrastructure/regions_az/

Google Cloud Architecture Framework. (2023, March 28). https://cloud.google.com/architecture/framework

Hallmans, D., Sandström, K., Nolte, T., & Larsson, S. (2015). Challenges and opportunities when introducing cloud computing into embedded systems. *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, 454–459. https://doi.org/10.1109/INDIN.2015.7281777

Hernandez, I., & Cole, M. (2007). Reliable DAG scheduling on grids with rewinding and migration. *1st International ICST Conference on Networks for Grid Applications*.

Hosseini, S. M., & Arani, M. G. (2015). Fault-tolerance techniques in cloud storage: A survey. *International Journal of Database Theory and Application*, *8*(4), 183–190.

Jakovljevic, M., Insaurralde, C. C., & Ademaj, A. (2014). Embedded cloud computing for critical systems. *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, 4A5-1-4A5-9. https://doi.org/10.1109/DASC.2014.6979465

Knight, J. C. (2002). Safety critical systems: Challenges and directions. *Proceedings of the 24th International Conference on Software Engineering*, 547–550. https://doi.org/10.1145/581339.581406

Kwon, Y., Balazinska, M., & Greenberg, A. (2008). Fault-tolerant stream processing using a distributed, replicated file system. *Proceedings of the VLDB Endowment*, *1*(1), 574–585. https://doi.org/10.14778/1453856.1453920

Lyon, B. K., & Popov, G. (2021). "What-if" Analysis Methods. In *Risk Assessment: A Practical Guide to Assessing Operational Risks* (pp. 137–151). John Wiley & Sons, Inc. Hoboken, NJ, USA.

Malaska, T., & Seidman, J. (2018). *Foundations for architecting data solutions: Managing successful data projects*. O'Reilly Media.

Marwedel, P. (2021). *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer International Publishing. https://doi.org/10.1007/978-3-030-60910-8

Mell, P., Grance, T., & others. (2011). *The NIST definition of cloud computing*.

Mesbahi, M. R., Rahmani, A. M., & Hosseinzadeh, M. (2018). Reliability and high availability in cloud computing environments: A reference roadmap. *Human-Centric Computing and Information Sciences*, *8*(1), 20. https://doi.org/10.1186/s13673-018-0143-8

Microsoft Azure Well-Architected Framework. (2023, March 28). https://learn.microsoft.com/en-us/azure/well-architected/

Mittal, D., & Agarwal, N. (2015). A review paper on Fault Tolerance in Cloud Computing. *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, 31–34.

Mukwevho, M. A., & Celik, T. (2021). Toward a Smart Cloud: A Review of Fault-Tolerance Methods in Cloud Systems. *IEEE Transactions on Services Computing*, *14*(2), 589–605. https://doi.org/10.1109/TSC.2018.2816644

Mulder, J. (2020). *Multi-Cloud Architecture and Governance: Leverage Azure, AWS, GCP, and VMware vSphere to build effective multi-cloud solutions*. Packt Publishing Ltd.

Newman, S. (2021). *Building microservices*. O'Reilly Media, Inc.

Patra, P. K., Singh, H., & Singh, G. (2013). Fault tolerance techniques and comparative implementation in cloud computing. *International Journal of Computer Applications*, *64*(14).

Prathiba, S., & Sowvarnica, S. (2017). Survey of failures and fault tolerance in cloud. *2017 2nd International Conference on Computing and Communications Technologies (ICCCT)*, 169–172. https://doi.org/10.1109/ICCCT2.2017.7972271

Prodan, R., & Ostermann, S. (2009). A survey and taxonomy of infrastructure as a service and web hosting cloud providers. *2009 10th IEEE/ACM International Conference on Grid Computing*, 17–25. https://doi.org/10.1109/GRID.2009.5353074

Regions and zones | Compute Engine Documentation. (2023, March 29). Google Cloud. https://cloud.google.com/compute/docs/regions-zones

Rehman, A. U., Aguiar, R. L., & Barraca, J. P. (2022). Fault-Tolerance in the Scope of Cloud Computing. *IEEE Access*, *10*, 63422–63441. https://doi.org/10.1109/ACCESS.2022.3182211

Ryan, A. (2010). *Formal specification of moving block railway interlocking using CASL*.

Service Level Agreement for Microsoft Online Services. (2023, June 1). https://view.officeapps.live.com/op/view.aspx?src=https%3A%2F%2Fwwlpdocumentsearch.blob.core.windows.net%2Fprodv2%2FOnlineSvcsConsolidatedSLA(WW)(English)(June2023)(CR).docx%3Fsv%3D2020-08-04%26se%3D2123-06-08T11%3A06%3A23Z%26sr%3Db%26sp%3Dr%26sig%3DCr7UJVzjBJ%252BTpWIbBdoDQJKe0hKCPdzbsOepuS%252BM%252B%252Fk%253D&wdOrigin=BROWSELINK

Surbiryala, J., & Rong, C. (2019). Cloud Computing: History and Overview. *2019 IEEE Cloud Summit*, 1–7. https://doi.org/10.1109/CloudSummit47114.2019.00007

Tischler, P. (2021). *Building Reliable Services on the Cloud*. O'Reilly Media, Inc.

Wang, J., Cho, J., Lee, S., & Ma, T. (2011). Real time services for future cloud computing enabled vehicle networks. *2011 International Conference on Wireless Communications and Signal Processing (WCSP)*, 1–5. https://doi.org/10.1109/WCSP.2011.6096957

Watchdog timer. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Watchdog_timer&oldid=1136797554

What are Azure regions and availability zones? (2023, March 15). https://learn.microsoft.com/en-us/azure/reliability/availability-zones-overview

Zen, K., Mohanan, S., Tarmizi, S., Annuar, N., & Sama, N. U. (2022). Latency Analysis of Cloud Infrastructure for Time-Critical IoT Use Cases. *2022 Applied Informatics International Conference (AiIC)*, 111–116. https://doi.org/10.1109/AiIC54368.2022.9914601

8 Appendix

8.1 SWIFT Guidewords

Categories	Guidewords	Description
Availability	• Uptime • SLA (Service Level Agreement) • Load Balancing • Auto Scaling • Disaster Recovery	This guideword refers to the ability of the application to always remain accessible and functional to users. Potential issues related to availability include downtime, service disruptions, and network connectivity issues.
Performance	• Resource Utilization • Application Tuning • Query Optimization • Caching	This guideword refers to the speed, responsiveness, and overall performance of the application. Potential performance issues include slow response times, resource

	• Indexing	bottlenecks, and inefficient code, algorithm, or inability to handle large volumes of traffic.
Data-Consistency	• Replication • Consensus • Atomicity • Isolation • Durability	This guideword refers to the accuracy and integrity of data that is stored and accessed by the application. Potential issues related to data consistency include data corruption, data duplication, and inconsistencies between different data sources.
Data Loss	• Disaster Recovery • Redundancy • Failover • Backup Frequency • Backup Retention	This guideword refers to the risk of losing important data that is stored or processed by the application. Potential causes of data loss include hardware failures, software bugs, user errors, and cyber-attacks.
Latency	• Network Optimization • Content Delivery Network (CDN) • Edge Computing • Application Architecture • Database Design	This guideword refers to the delay between the user's request and the application's response. Potential latency issues include network congestion, server overload, and inefficient data transfer protocols.
Scalability	• Auto Scaling • Load Balancing • Resource Utilization • Horizontal Scaling • Vertical Scaling	This guideword refers to the ability of the application to handle increasing volumes of traffic and data without compromising performance or stability. Potential scalability issues include slow response times, server overloading, and database capacity limitations.
Backups	• Backup Frequency • Backup Retention • Backup Validation • Backup Encryption • Backup Storage Location	This guideword refers to the regular creation and maintenance of backup copies of the application and its data, which can be used to recover from data loss or other issues. Potential issues related to backups include backup failures, incomplete backups, and outdated backup data.
Cost	• Resource Optimization • Cost Analysis • Instance Type Selection • Reserved Instances	This guideword refers to the financial costs associated with the development, deployment, and maintenance of the application. Potential cost issues include budget overruns, inefficient resource

	• Spot Instances	allocation, and unexpected expenses during development or maintenance.
Security	• Encryption • Access Control • Vulnerability Management • Audit Logging • Compliance	This guideword refers to the protection of the application and its data from unauthorized access, theft, or damage. Potential security risks include data breaches, malware infections, and unauthorized access by hackers or malicious insiders.
Network Security	• Firewalls • Access Control Lists • VPNs (Virtual Private Networks) • DNS (Domain Name System) Protection • IDS/IPS (Intrusion Detection and Prevention Systems)	This guideword refers to the protection of the application and its data from network-based attacks, such as DDoS attacks, man-in-the-middle attacks, or data interception. Potential network security issues include data breaches, service disruptions, and unauthorized access to sensitive data.
Compliance	• Regulatory Requirements • Audit Logging • Access Control • Data Protection • Incident Response	This guideword refers to the application's adherence to relevant industry standards, regulations, and policies, such as GDPR or HIPAA. Potential compliance issues include data privacy violations, regulatory penalties, and reputational damage.
Cloud service integration	• API Design • Interoperability • Data Transformation • Protocol Compatibility • Dependency Management	This guideword refers to the ability of the application to integrate with other cloud services and platforms, such as storage, messaging, or analytics services. Potential issues related to cloud service integration include incompatible APIs, data security risks, and performance bottlenecks.
Cloud provider lock in	• Interoperability • API Design • Data Portability • Service Availability • Vendor Relationship Management	This guideword refers to the potential difficulty of migrating the application to a different cloud provider or platform. Potential issues related to cloud provider lock-in include limited vendor choice, reduced flexibility, and dependency on proprietary cloud technologies.
Elasticity	• Resource Provisioning • Scaling Policies • Performance Metrics	This guideword refers to the ability of the application to dynamically provision and de-provision resources in response to changes in demand. Potential issues related to

	• Cloud Provider Limits • Cost Optimization • Resource Utilization	elasticity include resource contention, inefficient resource utilization, and excessive cloud provider costs.
Multi-tenancy	• Isolation • Tenant Management • Resource Allocation • Security • Performance	This guideword refers to the ability of the application to serve multiple users or tenants while maintaining data privacy and security. Potential issues related to multi-tenancy include data leakage, unauthorized access to tenant data, and performance degradation due to resource contention.
Disaster recovery	• Business Continuity • Recovery Point Objective (RPO) • Recovery Time Objective (RTO) • Backup Validation • Redundancy • Disaster Recovery Plan • Backup Testing	This guideword refers to the ability of the application to recover from a catastrophic event, such as a natural disaster or a cyber-attack, and restore normal operations as quickly as possible. Potential issues related to disaster recovery include incomplete data backup, insufficient failover capacity, and inadequate testing of recovery procedures.

8.2 “What if…” / “How could…” Questions

"What if…" / "How could…" Questions	**Recommended Action**	**Proposed Design Pattern**
What if there is a lack of isolation between different software components	Enforce logical and physical separation between different part of the systems	Critical Enclave
What if resource contention occurs between components, leading to performance degradation or resource monopolization?	Implement resource allocation and prioritization mechanisms to ensure fair sharing of resources among components.	Critical Enclave, Publish–Subscribe Pattern
What if the application experiences high latency due to frequent database queries	Utilize caching to optimize the retrieval of frequently accessed data and computation results.	Critical Enclave, Data Segmentation,

or resource-intensive operations?	Separate read/write use-cases, use read-only replica for reading.	Publish–Subscribe Pattern
What if the application experiences scalability limitations or performance bottlenecks?	Evaluate and optimize the application architecture for horizontal scalability and graceful degradation. Utilize microservices, containers, or serverless architectures to enhance scalability and performance.	Stateless Computation
What if sudden spikes in user traffic occur, overwhelming the application's resources?	Implement auto scaling to dynamically add more resources (e.g., virtual machines) based on demand.	Auto-Scaling, Stateless Computation
How could software be updated?	Implement redundant systems and graceful shutdown.	Stateless Computation, Multi-Availability Zone
What if a hardware or component failure occurs, causing service disruption?	Implement redundancy to ensure high availability.	Multi-Availability Zone
What if the computer system running the software system goes down unexpectedly?	Employ redundant systems and failover mechanisms as proactive measures to mitigate downtime and ensure continuous operation.	Multi-Availability Zone
What if the application experiences uneven traffic distribution, leading to performance issues?	Implement load balancing mechanisms to evenly distribute incoming traffic across multiple instances.	Multi-Availability Zone
What if a load balancer becomes a single point of failure?	Implement redundant load balancers to ensure high availability / rely on cloud provider solutions.	Multi-Availability Zone, Multi-Region Deployment
What if a disaster or service disruption occurs, impacting the availability of critical systems or data?	Implement redundant infrastructure	Multi-Region Deployment, Geo-Replication
What if a natural disaster or catastrophic event impacts the availability of cloud infrastructure?	Establish a geographically separate secondary setup in a different region or location. Replicate critical data and systems to	Multi-Region Deployment, Geo-Replication

	the secondary region for quick failover in case of a disaster.	
What if there is a prolonged outage of the cloud service due to equipment failure or infrastructure issues?	Establish the same setup in a different cloud provider	Multi Cloud Deployment
What if the application's resource utilization exceeds capacity, leading to performance degradation?	Establish the same setup in a different cloud provider	Multi-Region Deployment, Multi Cloud Deployment
What if there is a prolonged interruption to business operations due to a catastrophic event?	Replicate critical systems and data in geographically separate locations.	Multi-Region Deployment, Multi Cloud Deployment
What if the allocated resources are insufficient to handle the workload demand?	Implement automated resource provisioning mechanisms, such as auto scaling or dynamic resource allocation, to scale resources up or down based on workload requirements.	Multi-Availability Zone, Multi-Region Deployment, Multi Cloud Deployment
What if a transaction fails midway, leading to inconsistent data state?	Implement retry mechanism to reduce the propagation of potential failures.	API Gateway
What if the application's response time is slow or inconsistent?	Implement timeout and retry mechanism to reduce the propagation of potential failures.	API Gateway
What if the system cannot achieve strong consistency in a multi-region setup?	Implement workload distribution between regions. Use eventual consistency to deal with region failures.	Elastic Workload Segmentation, Geo-Replication
What if the components deliver a faulty response, such as bit-flip?	Ensuring the integrity of the data by adding payload signature. N-Version-programming to deploy multiple independent versions of components.	N-Version Programming and Deployment

What if the application's current infrastructure cannot handle increased user load?	Implement horizontal scaling.	Auto-scaling
What if data consistency or synchronization issues arise with horizontally scaled components?	Implement distributed caching mechanisms or messaging systems to ensure data consistency across horizontally scaled instances. Consider using distributed databases or replication strategies to handle data synchronization.	Geo-Replication
What if DNS services become unavailable?	Implement redundant DNS servers and utilize DNS anycast routing to distribute DNS requests across multiple locations.	Redundant DNS

8.3 Moving Block Fault Tree Analysis

The suffix R-A, refers to Region A, whereas suffix R-B refers to Region B.

Name	Description	Type	Logical Condition	Event Calculation Method	Failure Rate Type	Input Value 1	Required Inputs	Fault Tree ID	Parent ID	Parent Name	Gate / Event Indenture
Moving Block	Moving Block System	AND Gate	Normal	Constant	Failure	0,000000	0,000000	1478177			
Region A	Regional Outage	OR Gate	Normal	Constant	Failure	0,000000	0,000000	1478181	1478177	Moving	> Region A
DB1: Outage(R-A)	DB: Granted	Basic Event	Normal	Constant	Failure	0,001000	0,000000	1478186	1478181	Region A	> Region A > DB1: Outage(R-A)
DB2: Outage(R-A)	DB: Tracks & Zones	Basic Event	Normal	Constant	Failure	0,001000	0,000000	1478185	1478181	Region A	> Region A > DB2: Outage(R-A)
Messaging	Messaging Middleware	Basic Event	Normal	Constant	Failure	0,100000	0,000000	1478187	1478181	Region A	> Region A > Messaging Middleware(R-A)
Load Balancer(R-A)	Load Balancer Outage	Basic Event	Normal	Constant	Failure	0,010000	0,000000	1478189	1478181	Region A	> Region A > Load Balancer(R-A)
Compute(R-A)	Compute outage	OR Gate	Normal	Constant	Failure	0,000000	0,000000	1478190	1478181	Region A	> Region A > Compute(R-A)
Voter /Orch (R-A)	Voter/Orchestrator	Basic Event	Normal	Constant	Failure	0,010000	0,000000	1478193	1478190	Compute(R-	> Region A > Compute(R-A) > Voter /Orch (R-A)
Moving Block	Compute outage of at	Voting	Normal	Constant	Failure	0,000000	2,000000	1478192	1478190	Compute(R-	> Region A > Compute(R-A) > Moving Block App(R-A)
App-V1(R-A)	Moving Block App V1 -	Basic Event	Normal	Constant	Failure	0,010000	0,000000	1478204	1478192	Moving	> Region A > Compute(R-A) > Moving Block App(R-A) > App-V1(R-A)
App-V2(R-A)	Moving Block App V2 -	Basic Event	Normal	Constant	Failure	0,010000	0,000000	1478203	1478192	Moving	> Region A > Compute(R-A) > Moving Block App(R-A) > App-V2(R-A)
App-V3(R-A)	Moving Block App V3 -	Basic Event	Normal	Constant	Failure	0,010000	0,000000	1478202	1478192	Moving	> Region A > Compute(R-A) > Moving Block App(R-A) > App-V3(R-A)
Config Serv.(R-A)	Config Server -	Basic Event	Normal	Constant	Failure	0,010000	0,000000	1478191	1478190	Compute(R-	> Region A > Compute(R-A) > Config Serv.(R-A)
Region B	Regional Outage	OR Gate	Normal	Constant	Failure	0,000000	0,000000	1478180	1478177	Moving	> Region B
DB1: Outage(R-B)	DB: Granted	Basic Event	Normal	Constant	Failure	0,001000	0,000000	1478205	1478180	Region B	> Region B > DB1: Outage(R-B)
DB2: Outage(R-B)	DB: Tracks & Zones	Basic Event	Normal	Constant	Failure	0,001000	0,000000	1478235	1478180	Region B	> Region B > DB2: Outage(R-B)
Messaging	Messaging Middleware	Basic Event	Normal	Constant	Failure	0,100000	0,000000	1478234	1478180	Region B	> Region B > Messaging Middleware(R-B)
Load Balancer	Load Balancer Outage	Basic Event	Normal	Constant	Failure	0,010000	0,000000	1478233	1478180	Region B	> Region B > Load Balancer Outage(R-B)
Compute(R-B)	Compute Outage	OR Gate	Normal	Constant	Failure	0,000000	0,000000	1478232	1478180	Region B	> Region B > Compute(R-B)
Voter /Orch.(R-B)	Voter/Orchestrator	Basic Event	Normal	Constant	Failure	0,010000	0,000000	1478231	1478232	Compute(R-	> Region B > Compute(R-B) > Voter /Orch.(R-B)
Moving Block	Compute outage of at	Voting	Normal	Constant	Failure	0,000000	2,000000	1478230	1478232	Compute(R-	> Region B > Compute(R-B) > Moving Block App(R-B)
App-V1(R-B)	Moving Block App V1 -	Basic Event	Normal	Constant	Failure	0,010000	0,000000	1478228	1478230	Moving	> Region B > Compute(R-B) > Moving Block App(R-B) > App-V1(R-B)
App-V2(R-B)	Moving Block App V2 -	Basic Event	Normal	Constant	Failure	0,010000	0,000000	1478227	1478230	Moving	> Region B > Compute(R-B) > Moving Block App(R-B) > App-V2(R-B)
App-V3(R-B)	Moving Block App V3 -	Basic Event	Normal	Constant	Failure	0,010000	0,000000	1478226	1478230	Moving	> Region B > Compute(R-B) > Moving Block App(R-B) > App-V3(R-B)
Config Serv.(R-B)	Config Server -	Basic Event	Normal	Constant	Failure	0,010000	0,000000	1478229	1478232	Compute(R-	> Region B > Compute(R-B) > Config Serv.(R-B)

8.4 Moving Block Fault Tree Analysis Cut-Sets Probability >= 0.01

The suffix R-A, refers to Region A, whereas suffix R-B refers to Region B.

CUT SETS

Probability	Events
0.010000	Messaging Middleware(R-A), Messaging Middleware(R-B)

8.5 Moving Block Fault Tree Analysis Cut-Sets Probability >= 0.001

The suffix R-A, refers to Region A, whereas suffix R-B refers to Region B.

CUT SETS

Probability	Events
0.010000	Messaging Middleware(R-A), Messaging Middleware(R-B)
0.001000	Messaging Middleware(R-A), Load Balancer Outage(R-B)
0.001000	Messaging Middleware(R-A), Voter /Orch.(R-B)
0.001000	Messaging Middleware(R-A), Config Serv.(R-B)
0.001000	Load Balancer(R-A), Messaging Middleware(R-B)
0.001000	Messaging Middleware(R-B), Voter /Orch (R-A)
0.001000	Messaging Middleware(R-B), Config Serv.(R-A)