Architecture for distributed systems that facilitates a cloud-native AIOps implementations

Juan Pablo Ospina Herrera

Tesis de maestría presentada para optar al título de Magíster en Ingeniería

Director

Diego José Luis Botia Valderrama, Doctor (PhD) en Ingeniería Electrónica

Universidad de Antioquia

Facultad de Ingeniería

Maestría en Ingeniería

Medellín, Antioquia, Colombia

2024

Cita	Ospina Herrera [1]
Referencia Estilo IEEE (2020)	[1] J. P. Ospina Herrera, “Architecture for distributed systems that facilitates a cloud-native AIOps implementations”, Tesis de maestría, Maestría en Ingeniería, Universidad de Antioquia, Medellín, Antioquia, Colombia, 2024..

Maestría en Ingeniería, Cohorte XXXV.

Grupo de Investigación Intelligent Information Systems Lab..

Centro de Investigación Ambientales y de Ingeniería (CIA).

Centro de Documentación Ingeniería (CENDOI)

Repositorio Institucional: http://bibliotecadigital.udea.edu.co

Universidad de Antioquia - www.udea.edu.co

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKOWNLEDGEMENTS

I would like to express my heartfelt gratitude to my family for their unwavering support throughout my journey of pursuing a master's degree. Their constant encouragement, understanding, and sacrifices have been instrumental in my achievements. I am deeply grateful for their love, patience, and belief in me, which provided the motivation and strength to overcome challenges and strive for excellence. Their presence and belief in my abilities have been invaluable, and I am truly blessed to have such a loving and supportive family. Thank you for being my pillars of strength and for standing by me every step of the way.

GLOSSARY

1. **AIOps:** AIOps stands for Artificial Intelligence for IT Operations. It refers to the use of artificial intelligence and machine learning techniques to automate and enhance various aspects of IT operations, including monitoring, analytics, and incident management.
2. **DevOps:** DevOps is a software development methodology that combines software development (Dev) and IT operations (Ops) to improve collaboration, efficiency, and the quality of software delivery. It emphasizes automation, continuous integration, and continuous delivery to enable faster and more reliable software releases.
3. **Cloud-Native:** Cloud-Native refers to a software architecture and development approach specifically designed for cloud environments. It involves building applications that leverage cloud services, use containerization, and follow principles such as scalability, resilience, and elasticity.
4. **Distributed Systems:** Distributed Systems refer to computer systems composed of multiple autonomous computers or servers that communicate and coordinate their actions to achieve a common goal. They are designed to handle large-scale processing and storage requirements, providing fault tolerance and scalability.
5. **Software Architecture:** Software Architecture refers to the high-level design and organization of software systems. It defines the structure, components, relationships, and interactions between different software elements to achieve desired functionality, performance, and quality attributes.
6. **MTTD (Mean Time to Detect):** MTTD represents the average time it takes to detect an incident or an issue within a system. It measures the effectiveness and efficiency of monitoring and detection mechanisms.
7. **MTTR (Mean Time to Repair):** MTTR represents the average time required to repair or resolve an incident or failure in a system. It measures the responsiveness and efficiency of incident management and recovery processes.
8. **Non-functional requirements:** Non-functional requirements are the quality attributes or constraints that define how a software system should behave or perform. They include factors such as reliability, scalability, security, usability, and performance.
9. **Adaptability:** Adaptability refers to the ability of a software system to respond and adjust to changing requirements, environments, or conditions. It enables systems to handle evolving needs and external influences effectively.
10. **Interoperability:** Interoperability refers to the capability of different software systems or components to exchange and use information seamlessly. It ensures that disparate systems can work together and communicate effectively.
11. **Software pattern:** A software pattern is a reusable solution to a common software design problem. It provides a proven and documented approach to address specific challenges and improve the structure, flexibility, and maintainability of software systems.
12. **CI/CD (Continuous Integration/Continuous Delivery):** CI/CD is a software development practice that involves frequent integration of code changes and continuous delivery of software updates. It emphasizes automation, testing, and streamlined deployment processes to achieve faster and more reliable software releases.
13. **Docker:** Docker is an open-source containerization platform that allows developers to package and deploy applications along with their dependencies into lightweight, isolated containers. It provides consistency and portability across different environments.
14. **Kubernetes:** Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It provides features for load balancing, service discovery, and high availability.

15. **Microservices:** Microservices is an architectural style where applications are divided into small, independent services that can be developed, deployed, and scaled individually. Each service focuses on a specific business capability and communicates with other services via lightweight protocols.
16. **Serverless:** Serverless computing is a cloud computing model where developers can build and run applications without managing or provisioning underlying server infrastructure. It allows developers to focus solely on writing code and enables automatic scaling based on demand.
17. **Containers:** Containers are lightweight, isolated runtime environments that encapsulate applications and their dependencies. They provide consistency and portability by packaging software code, libraries, and configuration settings into a single deployable unit.
18. **C4 Model:** The C4 Model is a visual notation for describing software architecture at different levels of abstraction. It provides a set of diagrams that focus on Context, Containers, Components, and Classes to effectively communicate and document the architecture of a software system.
19. **AWS:** AWS, short for Amazon Web Services, is a comprehensive cloud computing platform provided by Amazon. It offers a wide range of cloud services, including computing power, storage, databases, networking, machine learning, and more, enabling organizations to build, deploy, and scale applications and infrastructure in the cloud.

1. ABSTRACT

The most popular approaches that have been used such as DevOps improve application operations thanks to the heyday of containers and CI/CD, however, this still requires human intervention in case of failures in any of the system components, since many of the solutions that have been used so far are limited to specific problems, such as reacting to downed servers and scaling them up. Taking into account that every time the operations of distributed systems become more and more complex due to the large number of components that must run at the same time, and also considering that in many applications, even small unavailability can translate into a strong impact on the reliability of the application, which implies an economic impact for the business, it is necessary that the solutions that are created reduce any type of risk and each time all these operations are more automated. Due to this, AIOps arises which uses artificial intelligence techniques such as machine learning and big data to operate and maintain application infrastructures, reduce the operational complexity of systems and automate IT operations processes. It has been proven that the implementation of this type of solution improves the quality of the systems and reduces the MTTD (Mean time to detect an error) from 10 minutes to 1 minute and the MTTR (Mean time to recovery) can be reduced from 60 minutes to 30 seconds [1], which is a very important advance in this world of operations. Despite this, there is still not so much adoption by most companies due to the challenges involved in implementing them in large projects and the fact that there is no clear path to follow when integrating applications to this type of solution that is emerging. In this research, we are going to propose a holistic architecture that makes it easier for cloud-native distributed systems to integrate with these new solutions.

KEYWORDS: AIOps, Software Architecture, Cloud-Native, Distributed Systems, Microservices, Serverless

2. PROBLEM STATEMENT

In distributed environments, maintaining multiple services running at the same time is challenging. AIOps seeks to reduce this operational complexity of the systems and automate all operational processes through the application of machine learning algorithms and big data to have services that are capable of managing themselves. Using AIOps, the MTTD can be reduced from 10 minutes to one minute with the help of AIOps and the MTTR can be reduced from 60 minutes to 30 seconds [1] which shows the great power that we can achieve by implementing this type of technology. With the wide variety of services that exist in the clouds today that provide different ways of operating and maintaining applications that are tightly coupled to the cloud of choice, a successful AIOps implementation that is easy to deploy in multiple scenarios becomes more complex.

The use of distributed systems implies several complexities such as the fact of using multiple programming languages, libraries, and frameworks, different development teams within the same system, the fact that each architecture carries its own challenges, and that there are multiple clouds that provide different customized services to manage the applications. All these complexities can be addressed by designing solutions that are as agnostic as possible for a specific tool, however, this also implies a challenge when defining the architecture that can be easily adapted to this type of application and that facilitates the implementation of AIOps.

Currently, there are customized solutions that focus on very specific tasks of IT operations management using artificial intelligence (focused on AIOps subcategories), however, there is no clear path to follow to make use of all these tools in such a way that has administration based on AIOps, applied especially to distributed systems where it is necessary to be aware of multiple services that run at the same time and with a component that fails, it can compromise the correct functioning of a part of the system.

Some previous research works only treat single tasks or subareas inside AIOps [9, 10, 11]. Most of them are mainly focused on the algorithm field [1], such as anomaly detection [12][13]-[14], clustering analysis [15], failure prediction [16][17]-[18], and cost optimization [19]. This motivates the need for a study focused on investigating the main architectures used in case studies of systems managed by AIOps solutions which is the reason for this research. We are going to apply some of the patterns identified in order to define a new architecture that will be a starting point for systems to migrate to this type of self-managed administration or new systems that want to take advantage of these powerful solutions.

3. STATE OF THE ART

A distributed system is a computing environment in which various components are spread across multiple computers (or other computing devices) on a network. These devices split up the work, coordinating their efforts to complete the job more efficiently than if a single device had been responsible for the task. Distributed systems are an important development for IT and computer science as an increasing number of related jobs are so massive and complex that it would be impossible for a single computer to handle them alone[1]. But distributed computing also offers additional advantages over traditional computing environments. Distributed systems reduce the risks involved with having a single point of failure, bolstering reliability and fault tolerance. Modern distributed systems are generally designed to be scalable in near real-time; also, you can spin up additional computing resources on the fly, increasing performance and further reducing time to completion. Today there are multiple examples of distributed architectures that are widely used in the software development industry, such as microservices and serverless architectures. These systems are strongly based on the concept of containerization, which is why the use of tools such as Docker or Kubernetes is very common.

Cloud-native computing is an approach in software development that utilizes cloud computing to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds[4]. Technologies such as containers, microservices, serverless functions, and immutable infrastructure, deployed via declarative code are common elements of this architectural style. These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil. Frequently, cloud-native applications are built as a set of microservices that run in Docker containers, and may be orchestrated in Kubernetes and managed and deployed using DevOps and Git CI workflows[4] (although there is a large amount of competing open source that supports cloud-native development). The advantage of using Docker containers is the ability to package all software needed to execute into one executable package. The container runs in a virtualized environment, which isolates the contained application from its environment.

Using the *IEEE Xplore*, *Scopus*, *ACM Digital Library*, *ScienceDirect*, and *arXiv* databases, an exhaustive search was made for papers that met all the inclusion/exclusion criteria. Different filters were used to discard articles based on their publication date between 2017-2023, English language, free online accessibility, and only articles or book chapters. The keyword set used for database searches is:

"aiops"
("implementation" OR "case study" OR "use case" OR "scenarios")
("architecture" OR "microservices" OR "serverless" OR "architectural patterns")
("distributed" OR "distributed systems")
("cloud" OR "cloud-native")

45 papers were selected, of which 39 were discarded, from which the relevant information for our research topic was extracted. We created a comparative table analyzing the relevant aspects and finally, we synthesized the main conclusions of this state of the art.

1. Evolving from Traditional Systems to AIOps: Design, Implementation and Measurements, 2020 [1]

In this paper, an AIOps system called Proton is created based on 5 key skills that the authors define: perception, detection, location, action, and interaction. Proton adopts a layered design with interoperability services between modules, making it highly compatible with traditional heterogeneous systems. It is implemented with some key considerations including data categories, database cluster, service gateway, and operation safety.

Tens of thousands of devices were used to test the AIOps system, of which three indicators are measured: the rate of self-repair of faults, the storage capacity, and the delay in the response of the service. In particular, the fault self-healing rate exceeds 80% for the scenario of server ping failure. The maximum write rate is approximately 80,000 metrics / second and the maximum query rate is approximately 7,500 metrics/second. And finally, it was obtained that the average response time is about 292 milliseconds.

To receive the HTTP requests, the option of using a service gateway was evaluated, however, this did not present the best results for the following reasons:

- Represents a bottleneck in high-performance network communication
- Became a possible single point of failure.
- Cannot achieve fine-grained permission control of the business, such as controlling the visibility of each data.
- The compatibility of the unified gateway is not guaranteed when using multiple frameworks.

Therefore, it was decided to use a DNS that had a load-balancing mechanism, while the service statistics are performed by each service. This strategy is applied to the Proton system and to the services that are integrated since both are connected in the last layer of the proposed design.

They also highlight the importance of capturing a large amount of information from each service, collecting data such as metrics, logging, tracing, configuration data, workflow data, and multimedia data. Each of this collected information is stored in a different database within the AIOps system.

Another important point of the article is security, for which two main aspects of the security mechanism were used: Authority and Approval of the operation (which is divided into single-point approval and global approval).

Finally, the authors mention the importance of using the microservices architecture, since interoperability can be increased by minimizing coupling using the restful protocol. In fact, many of these recommendations are inherent to this kind of architecture.

2. An Anomaly Detection Algorithm for Microservice Architecture Based on Robust Principal Component Analysis, 2019 [2]

In this paper, the authors propose an execution trace-based root cause location method for microservices architectures, which consists of two parts: Invocation chain anomaly analysis based on Robust Principal Component Analysis (RPCA) and a single indicator anomaly detection algorithm.

To test the algorithm, they used data from the AIOps 2020 International Challenge, which is an event that is planned to take place every year where issues related to AIOps are raised and where participants can validate their solutions against the data provided. The algorithm created

in this article scored 0.8304 (maximum score is 1) on the 4 batches of data used.

This challenge was based on a real problem presented in a commercial system with a microservices architecture. The data was provided by the Zhejiang Mobile company: static topology between services, the invocation chain data, the gold indicator data of the observed services, and the time-series data of the underlying services (operating system, Oracle, container, and middleware).

Although the challenge does not specify the complete architecture of the system, it is possible to identify that each microservice is implemented in a separate container and there is a middleware that receives the requests for each HTTP request. In addition, the importance they place on collecting information on the system's tracking and indicators is highlighted.

3. A Context Model for Holistic Monitoring and Management of Complex IT Environments, 2020 [3]

In this paper, the authors propose a holistic model named Monitoring Resource Model (MRM) for the management of context data, where it is important to know what context is required for a given monitored resource, where the context data are originating from, and how to access them across the data silos.

Based on this model, they propose a multilayer architecture consisting of 5 layers: acquisition, management, analysis, presentation of context data, and automated responses. The acquisition layer is responsible for communicating and collecting information from different data sources through adapters and performing the necessary transformations to the collected data.

The information that the model receives from managed applications is:

- UID: Unique identifier of the resource.
- KPIs: List of key performance indicators for this resource that are monitored by the respective monitoring systems and are used to describe the health status of a monitored resource. CPU load and percentage of memory used are examples of KPIs.
- Log data: Generated by each managed resource.
- Alerts: The alerts generated in the resource.

Furthermore, static information is collected which is not monitored, such as the IP or hypervisor of the virtual machine (VM) on which the resource is running. This information can be collected using management tools such as OpenStack and serves to detect the context in which the resource is running and thus be able to automate the creation of MRM instances of VMs.

At the end of the article, the authors analyze the main AIOps platforms currently available: Moogsoft, BMC, Splunk, and Broadcom, comparing the monitoring and management provided by each one, always thinking in a holistic approach, and comparing them with the MRM model proposed.

The authors also mention the rise of technologies such as Docker, Kubernetes, and OpenStack, which facilitate the creation of applications in the cloud, where additionally each cloud provider has its custom monitoring tools. The emerging AIOps platforms and the proposed MRM model are intended to work in conjunction with all of these technologies to facilitate the necessary management tasks.

4. AI-Governance and Levels of Automation for AIOps-supported System Administration, 2019 [4]

Although this paper does not focus on designing a custom AIOps solution or integrating a specific existing system with these platforms, the authors show the different levels of automation of application management, starting from 100% manual to the level that proposes AIOps where the platform is able to solve the errors by itself.

In the last section of the article, when discussing application scenarios, the authors mention that the main tasks of an AIOps platform are optimization of runtime, correction/recovery after a malfunction, updates, improvements of security, and sizing (scaling up / down). For each of these activities, the authors provide recommendations based on patterns that will facilitate the integration of the system with AIOps solutions.

For runtime optimizations, it is very useful to use virtualization and load-balancing tools. Additionally, they mention that typical activities such as scaling the compute nodes, replicating the database, and resizing the storage volumes, can impact the performance of the application but not the functionality or availability. Furthermore, measures to detect DDoS attacks such as reconfiguring the balancing frontends to block suspected IP addresses are one way to mitigate and prevent security problems.

The authors recommend using deployment tools like Kubernetes to manage updates, which allows gradual updates and has rollback mechanisms in case of any performance degradation.

Reliability, availability, and security issues can be automated with artificial intelligence strategies (unsupervised learning, deep learning, time series analysis), which rely heavily on the data generated by the application and the monitoring that is configured. They also highlight the importance of the system being self-healing and self-stabilizing, so that it is capable of generating a feasible remediation workflow autonomously.

5. Ananke: A framework for Cloud-Native Applications smart orchestration, 2020 [5]

The authors propose a framework called Ananke for monitoring and operating cloud-native microservices. They highlight the importance of modeling and monitoring CNA (cloud-native applications) to obtain data that we can use in our machine learning and big data models, facilitating integration with AIOps platforms. This information can be analyzed as time-series data, modeled as a time-varying multi-layer network. Companies like Splunk and Sysdig are looking to provide this type of monitoring.

The paper considers different communication protocols between microservices such as HTTP, gRPC, and Broker-based (asynchronous). It also takes advantage of the benefits of tools such as Kubernetes and OpenShift, performing online monitoring based on Prometheus architecture to minimize the interference of the observation. In addition, some popular patterns and principles in distributed architectures are highlighted, such as the fact that each service has its own database and API gateway as an entry point for requests for external clients.

The model that the article proposes allows representing the communication between the different microservices that can be presented with an external request, which can trigger a set of requests between the microservices to complete a transaction. This facilitates the creation of a time series graph and the entities that represent the model, therefore, the information on the performance of the entire application can be obtained by adding the data of the complete cluster

of all the requests.

Ananke's components are based on microservices patterns, which are divided into 3 main classes:

- Monitor and collect
- Storage and processing middleware
- Analyzers and actuators.

Collecting the time-series data is part of the monitoring layers that belong to the AIOps solution.

The authors also recommend creating libraries to facilitate code configuration quickly through decorators. Also, they suggest 2 other tools that they use within the Ananke framework: SPOUT and Raphtory.

Although the Ananke paper and framework are mainly focused on microservices architectures, they can be applied to different environments as function-as-a-service (serverless architectures) and metal-as-a-service, as the recommended patterns and principles are widely used in distributed systems.

6. Managing Distributed Cloud Applications and Infrastructure A Self-Optimising Approach. Chapter 3: Application Optimisation: Workload Prediction and Autonomous Autoscaling of Distributed Cloud Applications, 2020 [6]

In this chapter of the book, the authors discuss the optimization of the configuration and deployment of distributed cloud applications, which can be complex as it requires understanding factors such as the infrastructure and topology of the application, workload arrival and propagation patterns, and the predictability and variations of user behavior. For this, the chapter introduces the RECAP Application Optimization approach, which is a framework that enables the development and execution of optimization tasks at the application level.

The authors discuss optimization problems related to the placement, deployment, autoscaling, and remediation of applications. In addition, they show strategies to model these problems and with these models, predictions can be performed more accurately to support load balancing, autoscaling, and remediation proactively.

An important point mentioned by the authors is virtualization since it is widely used in the cloud and facilitates automation, scaling, optimization of resource allocation, and resilience to variations in workload intensity thanks to elasticity which is one of its main benefits. Elasticity requires load balancing that can generate additional parallel application components and redirect user requests to distribute the load evenly.

RECAP works for common architectures of distributed applications including client-server architecture, cloudlet, service-oriented architecture, and microservices. In these architectures, it is important to define the constraints of each component to model its communication patterns between them. Typically each component is split into the backend and frontend to allow each to scale independently. They also highlight the importance of collecting data from each component since this information is used by the models, however, this task is complicated in very large systems. The example presented at the end of the chapter shows an application that uses the RECAP framework. This application has a microservices architecture with a REST API Gateway that receives all requests from the system including the interaction it has with the

AIOps platform.

Analyzing the detail of each paper it is easy to identify certain similarities in the way the architectures of the reviewed applications are designed. Most of them are based on microservices, however many of the patterns and principles suggested in the articles can be applied in other architectural patterns of distributed systems such as Serverless.

A common topic among the applications reviewed in the papers is the approach to the cloud since all the proposed solutions are designed for cloud-native systems. Some authors propose layered models that allow the modeling of the AIOps platform and the application to be integrated, in order to clearly separate the responsibilities of each one. In these architectures, communication between layers is done using adapters that allow data to be processed in a way that is easier for machine learning and big data models.

TABLE 1
PATTERN, PRINCIPLES, AND TOOLS COMPARISON

ARTICLE / PATTERN	Architectural pattern	Cloud-Native	Patterns and principles	Tools
Evolving from Traditional Systems to AIOps: Design, Implementation and Measurements, 2020 [1]	Layered, Microservice	Y	Logging, tracing, Monitoring, Interoperability, DNS, Authority, Approval, REST	ES Cluster, MySQL, Hive, Influxdb, Python, Spring Cloud
A Context Model for Holistic Monitoring and Management of Complex IT Environments, 2020 [3]	Layered	Y	Monitoring, Virtualization	Docker, Kubernetes, OpenStack, AWS, MySQL
An Anomaly Detection Algorithm for Microservice Architecture Based on Robust Principal Component Analysis, 2019 [2]	Microservice, Reactive	Y	Synchronous communication, Middleware, Containerization, Logging, Monitoring	Docker, Oracle
AI-Governance and Levels of Automation for AIOps-supported System Administration, 2019 [4]	N/A	Y	Virtualization, Load balancing, CI/CD, Self-healing, Self-stabilizing	Kubernetes
Ananke: A framework for Cloud-Native Applications smart orchestration, 2020 [5]	Microservice, Serverless, Metal-as-a-service	Y	API gateway, REST, gRPC, Asynchronous communication, Monitoring, Analyzers, Actuators, Shared libraries	Prometheus, OpenShift, Kubernetes, Kafka, DBMS, SPOUT, Raphtory
Managing Distributed Cloud Applications and Infrastructure A Self-Optimising Approach. Chapter 3: Application Optimisation: Workload Prediction and Autonomous Autoscaling of Distributed Cloud Applications, 2020 [6]	Client-server, Cloudlet, Service-oriented, Microservices	Y	API gateway, REST, Load balancing, Autoscaling, Remediation, Virtualization, Monitoring	RECAP

In general, the top 3 recommendations to use when deploying applications alongside AIOps platforms are:

- Monitoring of all system components, relying on existing solutions provided by each cloud provider.
- A middleware in charge of intercepting requests from external clients and between services. Here there are multiple options such as using an API Gateway or a DNS that allows load balancing.
- Apply virtualization strategies for the deployment of components, especially based on containers using technologies such as Docker and Kubernetes as they provide additional advantages to scale each component independently.

Table 1 presents a comparison of the information found in each article focused on our research topic, summarizing the patterns, principles, and tools used in each paper. There we can see that all the reviewed solutions are focused on the cloud due to the benefits that facilitate the implementation in distributed systems. We found 3 recommendations that are common: monitoring of all components, middleware to intercept requests, and virtualization. Also, the most used architectures are microservices and layered, along with some very popular tools that facilitate working with distributed systems such as Kubernetes and Docker.

Now the question is which architectural pattern is better? To answer this question, we must first ask ourselves, which of these patterns makes it easier to integrate with the machine learning and big data algorithms used in AIOps? This requires an investigation where new architecture is defined and evaluated.

4. PROJECT OBJECTIVES

4.1. GENERAL

Propose a cloud-native architecture for distributed systems that facilitates integration with AIOps platforms, getting mean time to detect and repair errors metric in the standard time windows.

4.2. SPECIFICS

1. Evaluate the benefits and disadvantages of different architectural patterns and principles in applications that integrate with AIOps platforms to propose an architecture.
2. Implement an AIOps platform using cloud-native solutions.
3. Validate the easy integration of the proposed architecture with at least one specific use case, using different patterns, principles, and tools of distributed systems.

5. IMPACT AND PRODUCTS

With this work, we hope to establish the basis for applications to be more easily integrated with emerging AIOps solutions. This will benefit all those distributed projects where the administration of the different components has high operational complexity and any unavailability has a strong impact on the business side. Currently, research efforts on this topic are more focused on how to create different solutions for AIOps platforms, however, there is no clear path for existing or new applications to take full advantage of these solutions, this work presents a clear guide to the architectural patterns and principles best suited to AIOps solutions.

This work has an impact on the software development industry, as it shows the way for more projects to start integrating AIOps solutions and, as we described in the previous sections, the MTTD can be reduced from 10 minutes to 1 minute and the MTTR can be reduced from 60 minutes to 30 seconds. This translates into more robust and reliable applications, with a better reputation, and, in turn, has a good economic impact on the business.

As a result, we will deliver within the master's thesis a guide that will facilitate the implementation of an architecture for cloud-native distributed systems. In addition, we will publish a paper that will be sent to a journal recognized by Publindex and participate in an international conference to promote the findings.

6. METHODOLOGY

The methodology that will be applied for the development of the project is divided into three iterative stages, in which we will constantly verify that the results obtained are as expected. Figure 1 summarizes the 3 stages to follow.

First, we are going to perform an analysis of the different patterns, principles, and tools used in distributed systems, making comparisons of the advantages and disadvantages that each one provides. Then, an analysis of the benefits they have when applied together with an AIOps solution is proposed. With this, we will propose an architecture based on the analysis obtained, diagramming the different levels (using the C4 model) to have a holistic vision of the solution to be implemented.

After the architecture is defined, we are going to create an initial version of an AIOps platform using deep learning algorithms, combined with different existing development and deployment tools, it should contain the minimum functionalities to manage the operations of distributed applications. In this stage, we are going to combine some fixed solutions provided by each cloud provider with our custom big data and machine learning algorithms. This is with the aim of testing different algorithms to use. Additionally, we are going to use some public datasets with logs from distributed systems to train and test the implemented models. Then, we are going to choose the best models and the best services provided by the cloud providers to combine them and create the AIOps platform.

With all of the above, we proceed to test the proposed architecture with at least 2 controlled scenarios. At this point, as much information as possible should be collected from each test case in order to have more accurate results about the benefits of the proposed architecture. The 2 use cases will have small variations in the patterns and technologies to be used to validate that the architecture adapts correctly to different contexts. Here we are going to apply some cross-cutting patterns like the ones listed in section 8 (monitoring of all components, middleware to intercept requests, virtualization, and some others that we will identify in the first stage), this will allow us to reuse some components between the 2 use cases and will also allow us to have a more adaptable solution for any context. Finally, we are going to use a performance testing tool to run stress tests like JMeter to simulate cases where multiple users are making many requests on a system.

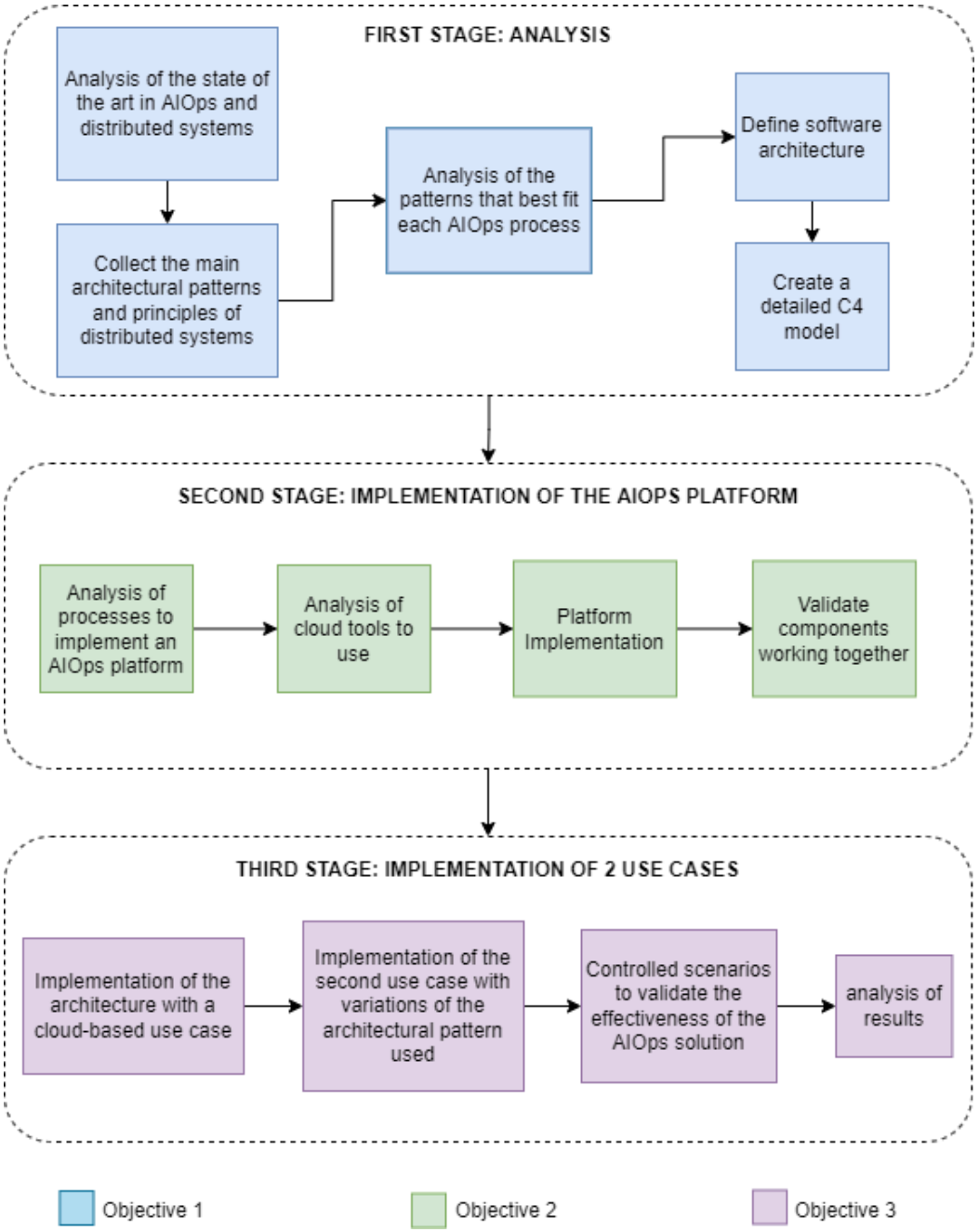

FIGURE 1. STAGED METHODOLOGY PROPOSED FOR PROJECT DEVELOPMENT

7. SCHEDULE

ACTIVITY/ MONTH	1	2	3	4	5	6	7	8	9	10	11	12	DELIVERABLE
1. Analysis of distributed architectures													
Analysis of the state of the art in AIOps and distributed systems													State of the art of AIOps and distributed systems
Collects the main architectural patterns and principles of distributed systems													List of patterns and principles
Comparison of patterns, analyzing advantages and disadvantages													Comparison chart
Analysis of the patterns that best fit each AIOps process													Reduced list with the best patterns
Definition of an architecture that facilitates the implementation of AIOps													C4 model
2. AIOps platform implementation													
Analysis of processes to implement an AIOps platform													List of steps to implement an AIOps platform
Analysis of cloud tools to use													List of possible cloud services
Platform Implementation													Software infrastructure configured in a specific cloud
Validate components working together													Report with test results
3. Validation of the architecture with 2 use cases													
Implementation of the architecture with a cloud-based use case													Software application
Implementation of the second use case with variations of the architectural pattern used													Software application
Controlled scenarios to validate the effectiveness of the													Report with test results

AIOps solution													
Analysis of results													Report with analysis of results
4. Presentation of the final report													
Preparation of the final report													Master's Thesis
Evaluation of the report													Evaluator feedback
Presentation of the results													Presentation

8. ANALYSIS OF DISTRIBUTED SYSTEMS PATTERNS

A distributed system is a network of independent computers that work together to provide users with a unified set of services. They are used to solve complex problems that would be difficult or impossible to tackle with a single, centralized computer. This type of system provides the ability to manage mission-critical services that are capable of scaling almost instantly in response to user demand, which is why it has become a necessity these days to build applications like distributed systems [21]. They offer several benefits over traditional centralized systems, such as better scalability, fault tolerance, and performance. However, they also pose challenges, such as ensuring consistency and reliability in the face of network failures, security threats, and communication delays. Building and maintaining distributed systems requires specialized knowledge and expertise. This involves designing and implementing protocols, algorithms, and architectures that are tailored to the characteristics and requirements of the application.

The first generation of distributed systems was based on the client-server model, where clients request services from a central server, which would respond with the requested information. This is considered the first step as it allowed for the frontend and backend to be independent and communicate through the network. More recently, microservices architecture has gained popularity as a way to build distributed systems, where applications are broken down into small, independent services that communicate with each other through APIs. This approach allows for greater flexibility and scalability, as services can be added or removed without affecting the entire system. Serverless computing is another recent trend in distributed systems, where applications are built and run on cloud platforms without the need for managing the underlying infrastructure [22].

The evolution of distributed systems has allowed the emergence of architectures such as microservices or serverless. They have evolved over time over time as new patterns are introduced or modified each day, so it is difficult to include them all in one list. For this research, the main patterns listed in books such as *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services* [21] and *Microservices Patterns* [23] are unified. The final list was limited to using patterns focused on monitoring components, middleware, and virtualization, as it was found that these points have a greater impact when integrated with AIOps tools during the state-of-the-art investigation. By integrating these areas with AIOps tools, it is possible to significantly improve the efficiency and effectiveness of system management, as these critical points allow identifying possible bottlenecks, failure points, and other performance issues.

8.1. MONITORING PATTERNS

In this section, I will compile all patterns related to application monitoring, including some patterns that facilitate observability. Application monitoring is essential for detecting issues and ensuring the smooth functioning of an application. It involves collecting and analyzing data on various aspects of the application, such as performance metrics, error rates, and user behavior, to identify potential problems and optimize the application's performance. Observability patterns can help developers and operators design applications that are more easily monitored and diagnosed, making it easier to identify and fix issues when they arise.

8.1.1. HEALTH CHECK API

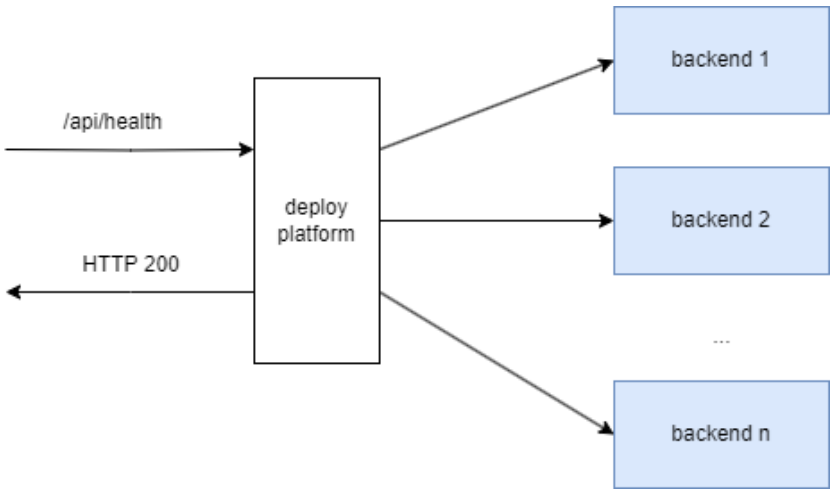

FIGURE 2. HEALTH CHECK API

The health check API pattern is a key pattern in microservices architecture that helps ensure the health and reliability of the services. As Chris Richardson describes in his book "Microservices Patterns" this pattern involves the implementation of an API endpoint that allows the microservice to report its current health status. The endpoint can be queried by an orchestrator or by other microservices to determine whether the service is currently healthy and available for use.

It is essential for maintaining the reliability and availability of distributed systems. By implementing a standard health check API, operators can easily monitor and manage the health of their services. The pattern also helps with fault tolerance and self-healing, as unhealthy services can be quickly identified and replaced [24]. Additionally, the health check API can be used to provide information on the dependencies of the service, such as databases or other microservices, and their health status. This can help to identify and resolve issues before they impact the system as a whole.

Implementing this pattern can be accomplished using a variety of tools and frameworks. One popular option is Spring Boot, which provides built-in support for health checks and can easily expose them through a REST endpoint. Another option is to use the Kubernetes liveness and readiness probes, which can be configured to execute custom health checks and report their status to the cluster. Cloud providers also offer tools to facilitate health checks, such as AWS Elastic Load Balancers and Azure Traffic Manager, which can monitor the health of instances and automatically route traffic to healthy ones.

8.1.2. LOG AGGREGATION

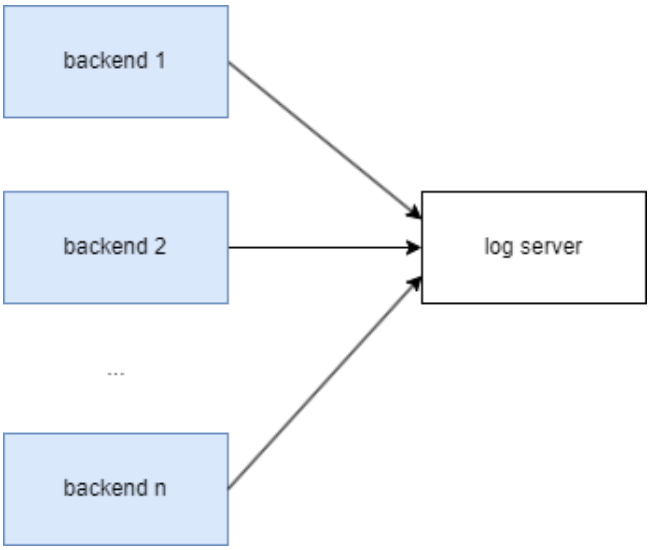

FIGURE 3. LOG AGGREGATION

The log aggregation is a pattern that involves collecting log data generated by distributed systems and applications and centralizing it for analysis and monitoring [25]. It is a common approach to address the challenges of distributed system logging. The pattern is useful for troubleshooting and identifying issues in distributed systems as it provides a holistic view of the system's behavior.

To implement log aggregation, various tools, and technologies can be used, such as Elasticsearch, Logstash, Kibana (ELK) stack, Fluentd, and Splunk. These tools provide the capability to ingest logs from different sources, transform them into a common format, and store them in a centralized location for analysis and visualization. Log aggregation can be a very effective means for detecting and diagnosing issues, as it enables analysts to search and explore a large amount of log data from a single place.

The log aggregation pattern is widely used in industry and academia, and many organizations have implemented it to monitor and analyze their distributed systems. For instance, Netflix uses the ELK stack for log aggregation to monitor and troubleshoot its microservices-based architecture.

8.1.3. DISTRIBUTED TRACING

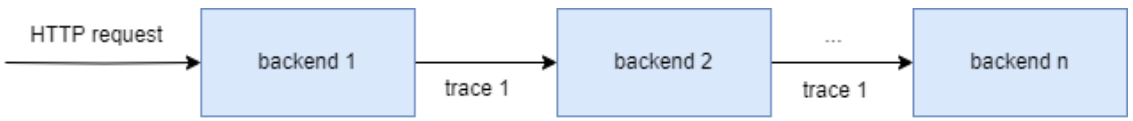

FIGURE 4. DISTRIBUTED TRACING

This is a pattern that is commonly used in cloud-native applications to gain visibility into complex distributed systems [27]. Distributed tracing involves the collection of timing and context data related to a specific user request as it travels through a distributed system. This data is then used to generate a trace of the request as it moves through different services, allowing developers to understand how the system is performing and to identify and diagnose problems.

One of the primary benefits of distributed tracing is that it provides a unified view of the entire system, making it easier to understand how different components are interacting with each other. This is particularly important in cloud-native applications, which are often composed of many services that are developed and deployed independently. With distributed tracing, developers can identify performance bottlenecks and errors that may not be immediately visible when looking at individual services in isolation.

Another important aspect of distributed tracing is the ability to correlate logs and metrics with individual requests. Logs and metrics are valuable sources of information when diagnosing issues, but they can be difficult to use in a distributed system where a single request may touch multiple services. Distributed tracing provides a way to tie these different sources of data together, making it easier to understand the context surrounding a particular request. It provides a way to allow developers to gain a better understanding of how different services are interacting with each other and how requests are flowing through the system.

Overall, distributed tracing is an important pattern for cloud-native applications that can help developers gain visibility into complex distributed systems. By providing a unified view of the entire system, correlating logs and metrics with individual requests [26], and enabling a better understanding of how services are interacting, distributed tracing is an invaluable tool for diagnosing issues and ensuring that cloud-native applications are operating at peak performance.

8.1.4. EXCEPTION TRACKING

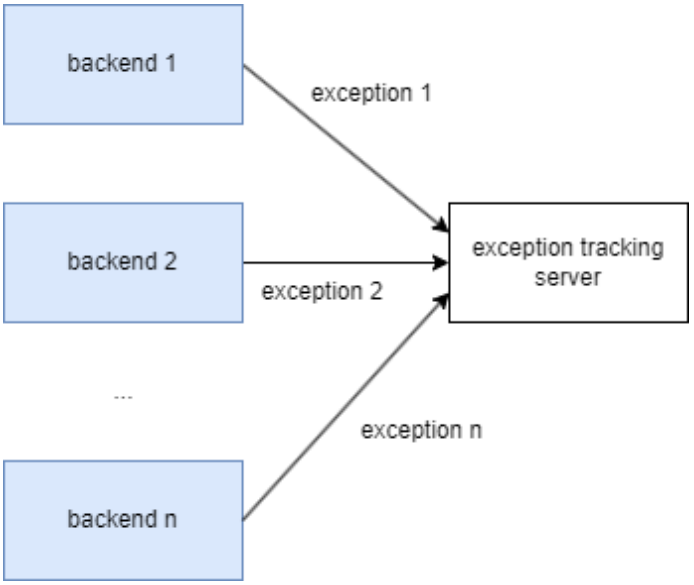

FIGURE 5. EXCEPTION TRACKING

The exception tracking pattern is a design pattern used in software development to identify, handle, and track exceptions that occur during the execution of a program [28]. Exceptions are runtime errors that can cause a program to crash, so it is important to have a mechanism to handle them gracefully. The pattern involves logging exceptions and capturing contextual information to aid in diagnosing and fixing the underlying issue. Exception tracking is crucial in the development of robust and reliable software, as it helps developers identify and resolve errors before they impact users.

In the context of AIOps, the exception-tracking pattern plays an important role in detecting and resolving issues in distributed systems. As modern applications become increasingly complex and distributed, it can be challenging to track down the source of an exception. AIOps platforms use machine learning algorithms to analyze large volumes of log data from across the system, enabling developers to quickly pinpoint the root cause of an issue. By leveraging the exception tracking pattern, AIOps platforms can help organizations reduce downtime and improve the overall reliability of their applications.

One approach to implementing the exception tracking pattern is to use cloud-native tools provided by major cloud providers such as AWS, Azure, and GCP. For example, AWS provides CloudWatch Logs and CloudWatch Alarms, which can be used to monitor application logs and trigger alarms based on predefined metrics. Azure provides Application Insights, which offers similar functionality, and GCP provides Stackdriver Logging and Error Reporting. These tools can be configured to capture application exceptions and log data, which can then be analyzed to identify patterns and trends. In addition, they can provide real-time alerts when exceptions occur, allowing DevOps teams to quickly respond and resolve issues.

Another option is to use third-party tools specifically designed for exception trackings, such as Sentry or Raygun. These tools provide more advanced features than cloud-native tools, such as the ability to group and filter errors by type or severity. They also offer integrations with other DevOps tools, such as Slack or PagerDuty, allowing teams to receive notifications and take action on errors more efficiently. However, these tools typically come with additional costs and may require more configuration and setup time than cloud-native solutions.

Regardless of the toolset chosen, it is important to implement a robust exception-tracking strategy in any cloud-native application. By identifying and resolving exceptions quickly, DevOps teams can ensure that their applications are performing optimally and that end-users are having a positive experience. Furthermore, by leveraging AIOps technologies such as machine learning and automation, exception tracking can become an even more powerful tool for ensuring application reliability and reducing downtime.

8.1.5. APPLICATION METRICS

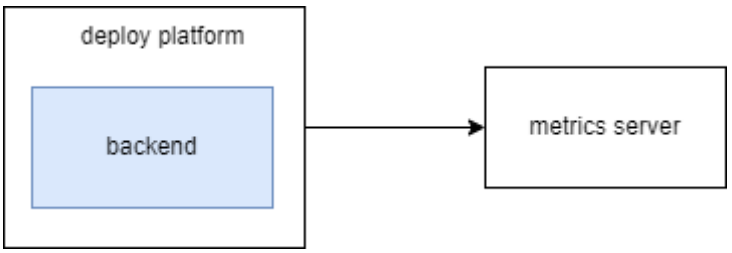

FIGURE 6. APPLICATION METRICS

In this pattern, the services report metrics to a central server that provides aggregation, visualization, and alerting [23] capabilities. There are multiple kinds of metrics to collect, such as CPU utilization, memory usage, disk utilization, request latency, and number of requests executed. Metrics can be collected at various levels of the application stack, including the infrastructure, platform, and application layers, and can be used to monitor everything from the health of individual services to the performance of an entire system. They are collected and stored in a central server, which provides visualization and alerting.

With the rise of cloud-native architectures and microservices, application metrics have become even more critical, as distributed systems can be complex and challenging to monitor. As a result, a range of tools and platforms have emerged to help organizations collect, store, and analyze application metrics, including popular solutions like Prometheus, Grafana, and Datadog. By leveraging these tools, organizations can gain real-time visibility into the performance of their applications, enabling them to quickly identify and address issues before they impact end users.

8.1.6. MICROSERVICE CHASSIS

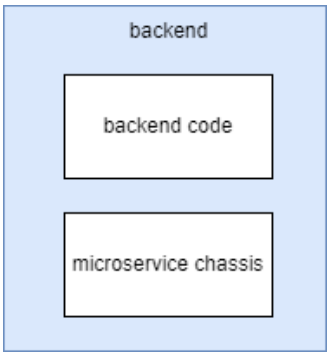

FIGURE 7. MICROSERVICE CHASSIS

The microservices chassis pattern is a pattern proposed by Chris Richardson that aims to provide a framework of shared infrastructure and libraries that handle cross-cutting concerns [23]. The idea is to extract common functionality such as service discovery, logging, metrics, and tracing into a separate framework that can be shared by all microservices. This pattern promotes the idea of building services that are lightweight, decoupled, and independently deployable. The microservices chassis pattern is not a monitoring pattern, but it can be used to provide observability features such as monitoring, logging, and tracing.

One of the key benefits of the microservices chassis pattern is that it provides a standardized way of implementing and managing observability features across multiple microservices. By having a common set of libraries and infrastructure, observability features can be implemented in a consistent and maintainable way. This can help to reduce the complexity and cost of implementing observability features across a large number of microservices.

The microservices chassis pattern can also help to improve the reliability and resiliency of services by providing features such as service discovery and circuit breaking. By providing a standard way of implementing these features, it can help to reduce the risk of errors and failures in microservices. It helps to improve the overall quality and performance of microservices, which is especially important in cloud-native applications.

8.1.7. SIDECAR

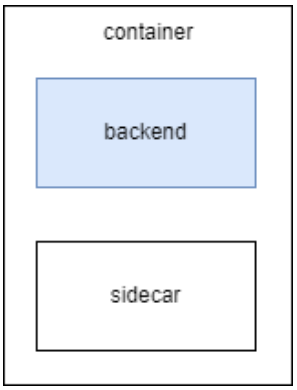

FIGURE 8. SIDECAR

The Sidecar pattern is a popular distributed system pattern that involves deploying a separate container alongside each instance to provide additional functionality without modifying the application code directly. This pattern enables the separation of concerns and helps to keep the services lightweight, simple, and decoupled. The sidecar container is responsible for providing features such as service discovery, load balancing, authentication, and monitoring, among others. By having a dedicated sidecar container, developers can easily update or replace these additional functionalities without affecting the core logic [21].

One of the main use cases for the Sidecar pattern is to enhance the observability of applications. The sidecar container can be used to collect and forward metrics, logs, and traces from the microservice to a central location for analysis and monitoring. This approach can simplify the implementation of observability features, as the sidecar container can be customized to handle the unique requirements of each instance. For example, a sidecar container can be used to collect metrics and logs for a specific microservice and forward them to a dedicated monitoring system, such as Prometheus or Grafana.

Another use case for the Sidecar pattern is to provide additional security features. By deploying a dedicated sidecar container, developers can add authentication and authorization features to the instance, and can also handle encryption and decryption of data in transit. This approach can help to improve the overall security of the architecture by providing a central point for security management. It can also be used to provide additional resilience features like circuit breakers, retries, and timeouts to the microservice without affecting its core functionality. The sidecar container can be configured to handle these resilience features and can automatically retry failed requests or stop sending requests to a failing service.

8.2. MIDDLEWARE PATTERNS

This section will gather all patterns related to intercepting HTTP requests in the backend using any kind of middleware. Intercepting HTTP requests is a technique that allows developers to modify requests before they are processed by the server. By intercepting requests, developers can add additional functionality to their applications without having to modify the client-side code. There are several patterns available for intercepting HTTP requests, each with its own strengths and weaknesses.

8.2.1. REPLICATED LOAD-BALANCED SERVICES

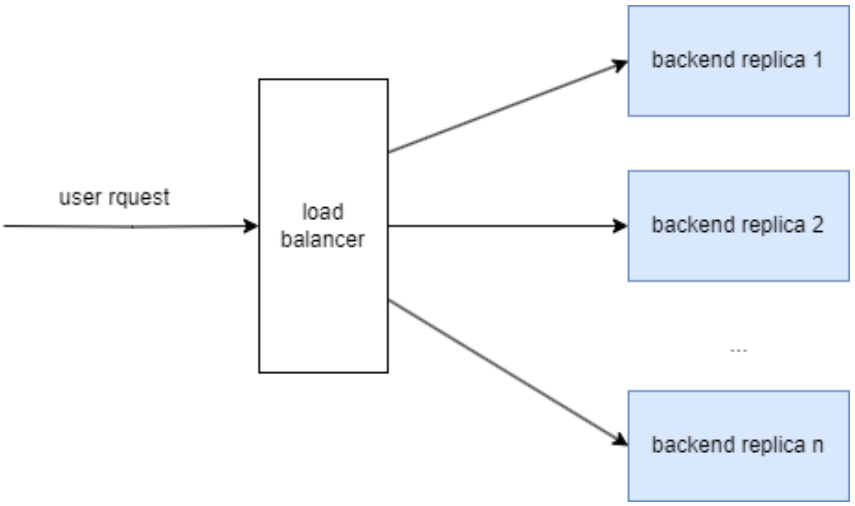

FIGURE 9. REPLICATED LOAD-BALANCED SERVICES

The replicated load balancer pattern is a distributed systems pattern that addresses the need for high availability and scalability in systems that require load balancing. The pattern consists of a scalable number of servers with a load balancer in front of them [21]. It increases the availability and traffic distribution across multiple instances to increase scalability. The nodes share a common configuration and synchronize the session state to provide consistent responses to clients.

This pattern is particularly useful in cloud-native applications that require load balancing across multiple instances of the same service or application. By replicating the load balancer component, the system is able to continue working even if one or more nodes fail. Additionally, by distributing traffic across multiple instances, the system is able to handle increased traffic without overloading any one instance.

The replicated load balancer pattern is based on a popular algorithm called the Round Robin. In this algorithm, incoming requests are distributed evenly across a set of backend servers [29]. Each incoming request is assigned to the next server in the list, ensuring that each server receives an equal share of the requests. However, there are other popular algorithms that can be used with this pattern, such as the Least Connection algorithm, where incoming requests are directed to the server with the least number of active connections, and the IP Hash algorithm, where the client's IP address is used to calculate which server to route the request to. The choice of algorithm depends on the specific requirements and characteristics of the application and infrastructure.

There are various tools and services available that can be used to implement the replicated load balancer pattern, including both open-source and commercial load balancers. For example, the NGINX Plus load balancer provides a clustering feature that allows for the replication of the load balancer component across multiple nodes. Additionally, cloud providers such as AWS, Azure, and GCP provide load-balancing services that can be used to implement the pattern.

8.2.2. OWNERSHIP ELECTION

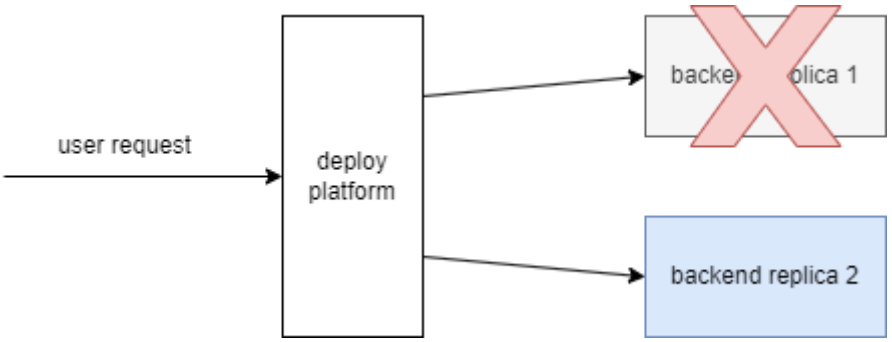

FIGURE 10. OWNERSHIP ELECTION

The ownership election pattern is a distributed systems pattern that addresses the problem of selecting a leader or primary node in a distributed system. In a distributed system, it is often necessary to elect a single node to perform a specific task or coordinate the actions of other nodes. This pattern provides a mechanism for electing a node to perform this role and ensuring that the node is the only one responsible for the task.

One common use case for the ownership election pattern is in the implementation of a distributed database system. In this scenario, it is necessary to select a primary node to handle write requests and coordinate data replication across the other nodes in the system. It provides a way to elect a primary node in a way that is fair and deterministic, ensuring that all nodes have a chance to serve as the primary.

One important consideration is the need to handle failure scenarios. If the primary node fails or becomes unavailable, it may be necessary to hold a new election to select a new primary [21]. This can be achieved using a variety of techniques, such as timeouts or heartbeat messages, to detect when a node has failed and trigger a new election.

It is considered a middleware because it acts as a mediator between different nodes in a distributed system, enabling them to communicate and coordinate the ownership of shared resources. By electing a leader among the nodes, the pattern establishes a central point of authority that can arbitrate access to shared resources. This helps to prevent conflicts that might arise if multiple nodes were trying to access the same resource simultaneously. It is a key solution that requires coordination and resource management between multiple nodes.

8.2.3. ADAPTER

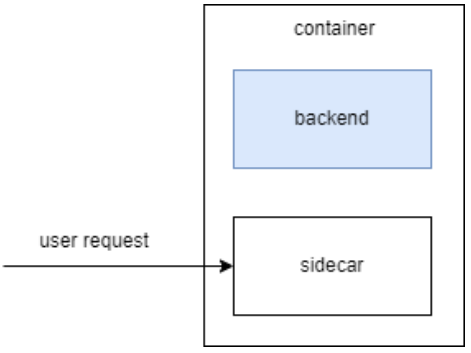

FIGURE 11. ADAPTER

This pattern is similar to the Sidecar, but now the adapter container is used to modify the interface of the application container so that it conforms to some predefined interface that is expected of all applications [21]. The adapter container is created to encapsulate the code that is responsible for translating messages between the different components. It acts as a middleware layer that sits between the two components and translates the messages that are sent between them.

The adapter container pattern is commonly used in distributed systems to help with service integration. Each service is designed to be small and independent, and it communicates with other services using lightweight protocols such as REST or messaging systems like Kafka or RabbitMQ. Each service is responsible for its own data and business logic, which means that it can be developed and deployed independently of other services. However, when different services need to communicate with each other, they may use different message formats or protocols. The adapter container pattern provides a way to bridge this gap by creating a container that is responsible for translating messages between different services.

8.2.4. ASYNCHRONOUS MESSAGING

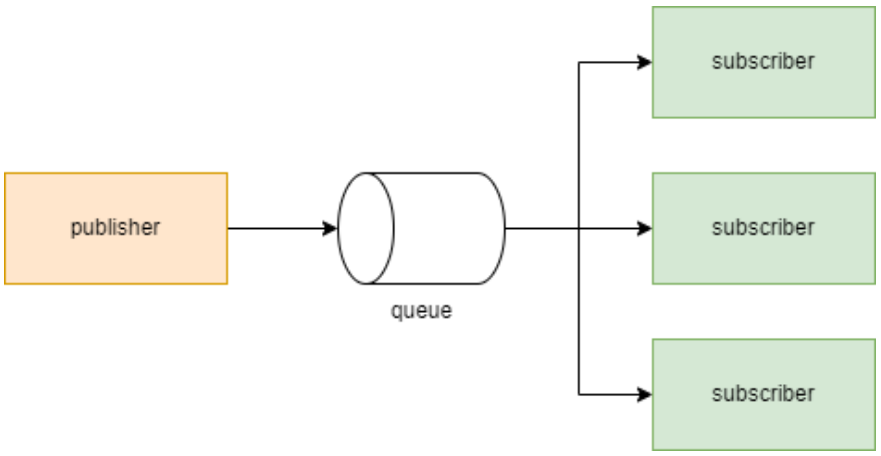

FIGURE 12. ASYNCHRONOUS MESSAGING

The Asynchronous messaging pattern is a communication pattern used in distributed systems that decouples the sender of a message from the receiver, allowing for more flexible and reliable communication. In this pattern, messages are sent asynchronously, meaning that the sender does not wait for a response from the receiver before continuing its operations [23]. Instead, the message is placed in a message queue, which acts as an intermediary between the sender and receiver. The receiver can then consume the message at its own pace, without any requirement for a real-time connection between the two parties.

Asynchronous messaging can be implemented in various ways, such as using message brokers or publish-subscribe systems. A message broker is a middleware component that acts as a hub for messages, allowing for the routing, transformation, and storage of messages [30]. Publish-subscribe systems, on the other hand, allow multiple consumers to receive the same message, providing a more scalable and fault-tolerant communication mechanism.

Asynchronous messaging provides several benefits, including increased scalability, reliability,

and flexibility. By decoupling the sender and receiver, systems can be designed to handle higher message volumes and provide fault tolerance, as messages can be stored and retried if a receiver is temporarily unavailable. This pattern is particularly useful for systems that require real-time data processing, such as financial transactions or real-time analytics.

However, implementing asynchronous messaging can also introduce complexity, as developers need to ensure that messages are processed correctly and in the intended order. Additionally, the use of message brokers and publish-subscribe systems can introduce latency and overhead, which can affect system performance. To mitigate these issues, careful consideration should be given to the design of the message format, the choice of message broker, and the implementation of the messaging client.

8.2.5. API GATEWAY

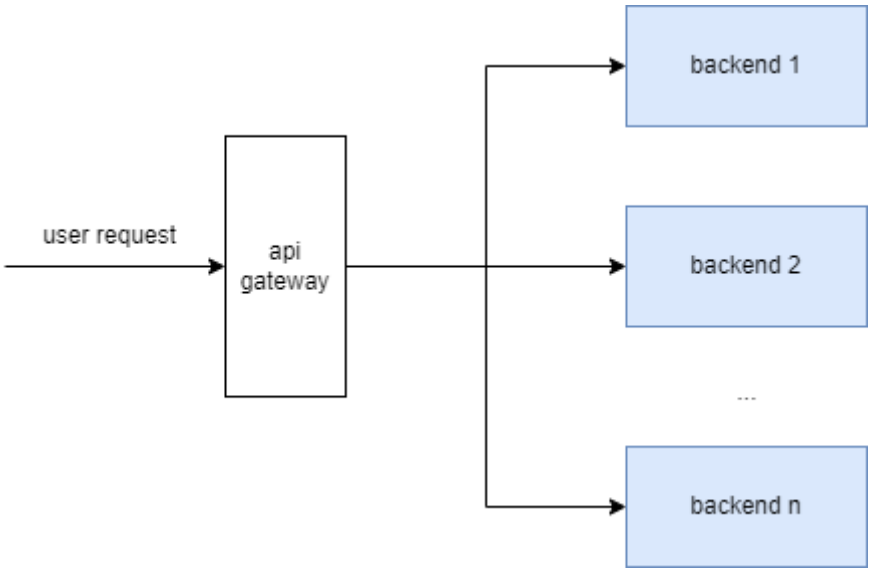

FIGURE 13. API GATEWAY

The API Gateway is a pattern that provides a centralized point of entry for client applications to access backend services in a distributed system. As a result, it can simplify the process of exposing multiple services to clients, especially those using different protocols or data formats. The API Gateway can also provide additional functionalities such as authentication, authorization, rate limiting, caching, and routing [32].

The API Gateway pattern can be implemented in various ways, including as a standalone component or as part of a larger service mesh architecture. Some cloud providers, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure, offer managed API Gateway services that can be easily configured and deployed.

One key benefit of using the API Gateway pattern is the ability to decouple clients from services [31]. This allows for greater flexibility in the deployment and scaling of services, as well as the ability to change or update services without affecting client applications. Additionally, the API Gateway can provide a layer of security and governance by enforcing access controls and monitoring traffic to backend services.

Another advantage of the API Gateway pattern is the ability to aggregate and consolidate data from multiple backend services. This can simplify the process of data integration for client applications and reduce the complexity of the overall system architecture. The API Gateway can also provide caching and rate-limiting features to improve performance and reliability.

However, the API Gateway pattern can also introduce additional complexity and potential points of failure in the system architecture. It requires careful design and configuration to ensure that it can handle high volumes of traffic and avoid creating bottlenecks [23].

To maximize the benefits of the API Gateway pattern, it is important to carefully consider the requirements and characteristics of the system architecture and choose the appropriate implementation and configuration options. This can involve selecting the appropriate protocols and data formats, defining access controls and authentication mechanisms, and configuring routing and caching policies.

Additionally, monitoring and observability are critical for ensuring the performance and reliability of the API Gateway and the overall system. This can involve using logging and metrics to track traffic and usage patterns, as well as identifying and diagnosing errors and issues.

8.2.6. SERVICE MESH

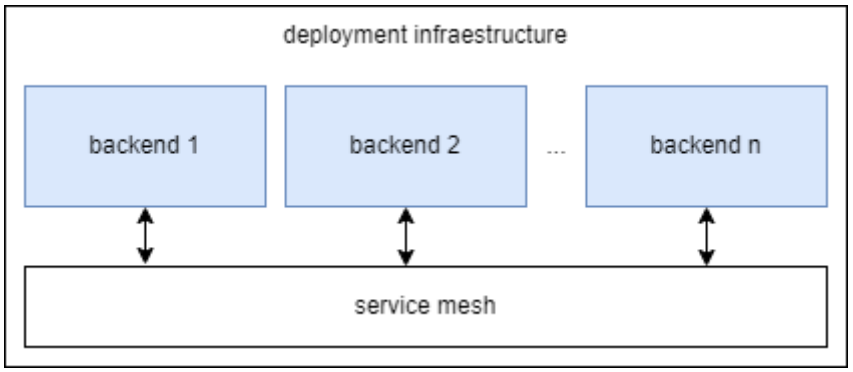

FIGURE 14. SERVICE MESH

The service mesh pattern is a modern approach to managing communication between services in a distributed system. It aims to provide a standardized way to route network traffic in and out of services through a networking layer that implements various concerns, including circuit breakers, service discovery, load balancing, traffic routing, security, and rule-based routing [23]. The pattern involves adding a dedicated infrastructure layer to the system, consisting of a set of interconnected network proxies that provide these functionalities.

The main benefit of using a Service Mesh is that it abstracts away the complexity of service-to-service communication from the application code, making it easier to develop, deploy, and manage. With other approaches like microservices chassis, you are restricted to the technology stack to use in the instances, with this pattern you can create this middleware in any programming language and framework and be used for all the instances.

One key feature of service meshes is the ability to implement traffic routing and load balancing

policies at the network layer, without requiring changes to the application code. This allows for more dynamic and flexible traffic management and can be especially useful in scenarios where services have varying levels of capacity or availability.

There are several popular service mesh implementations available today, including Istio, Linkerd, and Consul Connect. These implementations are typically implemented as a set of network proxies that intercept traffic between microservices and apply policies and rules to manage that traffic. They also provide enhanced security features, such as mutual TLS authentication, and can help simplify the management of security policies and access control.

One of the challenges of implementing a service mesh is the additional infrastructure required to support it. They can add significant overhead to a system, in terms of both compute resources and operational complexity. It requires careful planning and configuration to ensure that they are correctly integrated with the rest of the system.

Another challenge is the potential impact on performance and latency. Since service meshes add an additional layer of network proxies, they can introduce additional latency and reduce overall system performance [33]. However, many service mesh implementations have features to mitigate this impact, such as connection pooling and optimized network protocols.

Despite these challenges, this pattern has become increasingly popular in recent years, as organizations have adopted distributed system architectures and moved towards cloud-native development. It provides a powerful set of features for managing communication and observability in a distributed system and is likely to continue to play a key role in the development of modern, cloud-native applications.

8.3. VIRTUALIZATION PATTERNS

The virtualization of web applications involves running web applications in a virtualized environment, which allows multiple applications to run on a single physical machine. This technique is often used to improve the scalability, performance, and security of web applications. It is a common practice in the technology industry and can offer many benefits for companies looking to improve their infrastructure. In this section, I will compile all the patterns related to this.

8.3.1. VIRTUAL MACHINES

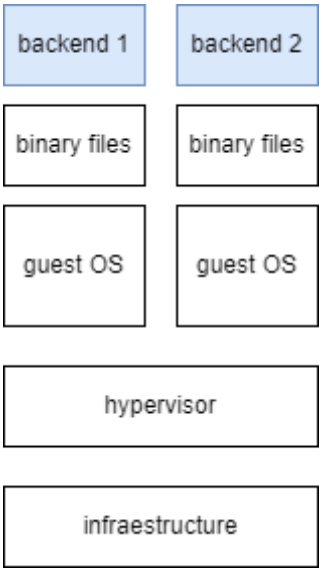

FIGURE 15. VIRTUAL MACHINES

This pattern consists on deploy services packaged as virtual machine images into production. Each service instance is an individual virtual machine [23] which helps isolate the service from other services running on the same server. Each VM can have its own operating system, software, and configuration, which allows for greater flexibility and control over the environment in which the service runs.

One of the main advantages of this pattern is the ability to scale services independently of each other. Each VM instance can be scaled up or down based on demand, allowing services to be deployed and managed in a more efficient manner. This approach also provides greater fault tolerance, as failures in one VM instance do not affect the availability of other services running on the same server.

Another benefit of this pattern is the ability to easily deploy and manage services across different cloud providers or on-premises environments. By packaging services as virtual machine images, they can be easily moved between different infrastructure environments without needing to reconfigure the entire system. This approach allows for greater flexibility and portability in the deployment of distributed systems.

However, there are some drawbacks to the VM per service pattern. One of the main concerns is the increased overhead and complexity associated with managing multiple VM instances [34]. Each instance requires its own operating system, software, and configuration, which can lead to higher costs and increased management complexity. Additionally, this approach can lead to longer deployment times, as each VM instance needs to be configured and deployed separately.

8.3.2. CONTAINERS

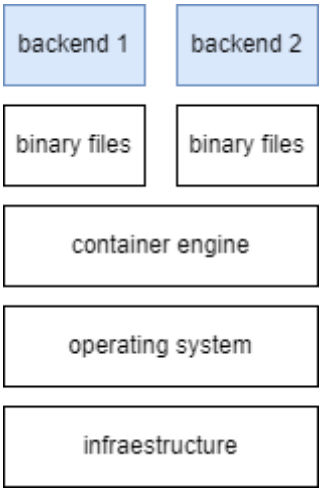

FIGURE 16. CONTAINERS

In recent years, containerization has emerged as an alternative to the VM per service pattern [23]. Containers provide a lightweight and portable way to package and deploy applications, without the overhead of a separate operating system for each service. This is the most popular deployment pattern used to deploy distributed systems into production. A container is a lightweight, stand-alone executable package that includes everything needed to run the service, including code, libraries, and system tools. Container technology, such as Docker, has revolutionized the way systems are deployed, making it easier and more efficient than ever before.

One of the key benefits of the container per service pattern is that it allows for each service instance to be isolated from the others. Each container has its own file system, network, and resources, which means that if one service instance fails or experiences issues, it will not impact the other instances. This pattern also enables easy scaling of services, as new instances can be deployed quickly and easily by simply creating a new container.

Container images can be stored in a registry for easy access and deployment. A container registry is a centralized place for storing and managing container images. It allows developers to push and pull container images as needed, making it easy to distribute applications across different environments. Container registries can be hosted in the cloud or on-premises, and they can be public or private, depending on the needs of the organization. Some examples of popular container registries include Docker Hub, Google Container Registry, and Amazon Elastic Container Registry. By using a container registry, organizations can simplify their deployment process and ensure consistency across their infrastructure.

8.3.3. KUBERNATES

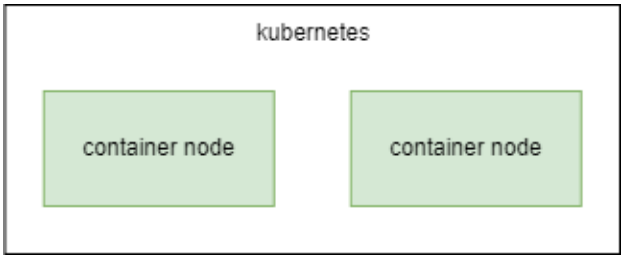

FIGURE 17. KUBERNATES

Using containerization without a container orchestration system like Kubernetes has several disadvantages. One of the main drawbacks is the lack of automation and scaling capabilities. Without a container orchestrator, developers are required to manually manage the deployment and scaling of containers. This can be time-consuming and prone to errors, especially as the number of containers and services grows. Another disadvantage is the lack of container networking capabilities. Without a container network overlay provided by Kubernetes, containers cannot communicate with each other outside of the host machine. This can make it difficult to build architectures that rely on service discovery and communication. In addition, without a container orchestration system, there is no centralized control plane to manage container workloads. This can lead to inefficient resource utilization, as it may be difficult to balance the allocation of resources across different containers and services.

Kubernetes uses a declarative approach to manage the state of the infrastructure, ensuring that the desired state is always maintained. It provides a wide range of features, such as automatic scaling, load balancing, rolling updates, and self-healing capabilities. These features make it easier for developers to deploy and manage their applications in a distributed environment. This pattern also allows developers to easily deploy and manage their applications across multiple environments, such as on-premises data centers, public clouds, and hybrid clouds.

This pattern provides all the advantages of containers and includes the benefits of having an orchestration system. This must be the default solution to use for deploying distributed systems in most cases, it has become the de-facto standard for managing containerized applications in the cloud. It would also be possible to use other self-managed approaches like serverless and get the same benefits, but the use cases for applying it are fewer than Kubernetes. Kubernetes includes tools for monitoring, logging, and tracing, as well as a wide range of third-party plugins and extensions.

Another benefit of this pattern is that it allows developers to easily scale their applications up or down as needed. Kubernetes can automatically scale the number of application instances based on resource utilization and demand, ensuring that the application can handle increased traffic or workloads. This makes it easier for developers to build applications that can handle unpredictable spikes in traffic or workloads, without having to worry about managing infrastructure resources manually.

Kubernetes has revolutionized the world of DevOps by providing a powerful platform for container orchestration and management [35]. DevOps teams can easily deploy and manage containerized applications across multiple environments, from development to production. It provides a consistent and standardized way to deploy and manage applications, making it easier to move workloads between different environments and platforms. Kubernetes also provides a powerful set of APIs and tools that enable DevOps teams to automate many tasks, such as scaling, rolling updates, and monitoring.

Kubernetes also provides a rich set of monitoring and logging capabilities that enable AIOps in distributed systems. It allows operators to monitor the health of the application and the underlying infrastructure by providing metrics and logs that can be easily aggregated and visualized. Kubernetes provides a standardized way to collect metrics from different components of the application and the infrastructure, which can be used to identify performance bottlenecks, diagnose issues, and improve the overall system performance.

9. ANALYSIS OF THE PATTERNS FOR AIOPS SOLUTIONS

After analyzing the list of patterns for each category, we can detect the similarity that exists among many of them. It is not uncommon to find similarities between patterns in software development. Developers often face similar problems and come up with similar solutions to solve them. However, some patterns may require more effort to implement than others. It's important to carefully evaluate each pattern and determine its feasibility and potential impact on the application. By doing so, developers can choose the most appropriate pattern for their specific needs and requirements. Although there are more patterns, the selected ones are those that have been identified as the most useful and effective in those areas, with which architectures for distributed systems can be easily built.

Some conclusions that we can highlight from these patterns are:

1. In the monitoring section, all patterns must be used, as the more information is collected about the state of the application, the better decisions can be made, especially when working with AIOps platforms where information is the most important. The use of monitoring patterns is crucial in collecting the right data and insights to help identify and resolve issues.
2. Health check API, log aggregation, distributed tracing, and exception tracking are patterns we can achieve with a bit of configuration. The last 3 will require external services to send the information.
3. The application metrics pattern requires more coding than previous ones. We will need to define in which places we need to collect data.
4. Microservice chassis and sidecars are useful patterns for applying reusable features. Actually, we could use just one of them and will get the same results.
5. For the middleware patterns, we can select only one of the last 2 and it will be enough. The best pattern may vary, but the API gateway is easy to implement than the service mesh pattern.
6. Replicated load-balanced services, ownership election, and adapters are patterns already implemented in tools like Kubernetes.
7. The asynchronous messaging pattern is only applicable for specific use cases.
8. The big winner in the virtualization section is Kubernetes. It contains all the features of the other 2 patterns and with extra features.

These conclusions favor the interests of the AIOps platforms, where we have a large amount of data from monitoring, a middleware with many features to offer, and a container orchestration tool that has superpowers for instance management. Based on this we can build the architecture for this research by combining these patterns and taking advantage of these conclusions.

TABLE 2
RECOMMENDED PATTERNS

SECTION	RECOMMENDED	DISCARDED
Monitoring	Health check API, log aggregation, distributed tracing, exception tracking, application metrics, microservice chassis, sidecar	
Middleware	API Gateway, Service Mesh, Asynchronous messaging	Replicated load-balanced services, Ownership election, and Adapters
Virtualization	Kubernetes	Virtual machines, containers

10. PROPOSED ARCHITECTURE - AIODS

Based on the previous analysis, a general architecture is proposed that is easy to adapt and implement by developers working with distributed systems. AIODS (AIOps for Distributed Systems) is an architecture focused on the integration between distributed systems with AIOps platforms. For this, the C4 model was used as a tool to create and visualize the different views of the architecture, this because it is a holistic way of diagramming a system from all its levels.

The C4 model is a software architecture model that provides a way to create and communicate software architecture diagrams in four views that are both easy to understand and effective. Each of these levels of the C4 model is focused on a certain perspective of the system [36]. It was created by Simon Brown, a software developer and consultant who wanted to create a simple yet powerful way to communicate software architecture.

The C4 model is based on a hierarchy of four levels: system context, container, component, and code. Each level provides a different level of detail and abstraction, allowing developers and architects to focus on the appropriate level of detail at each stage of the architecture design process.

At the highest level, the system context level provides a high-level view of the software system and its environment. This level is useful for communicating the system's purpose, scope, and stakeholders.

The container level provides a view of the system's containers and their interactions. Containers can be thought of as the runtime environments in which components are executed. This level is useful for understanding how the system is partitioned into deployable units.

The component level provides a view of the system's components and their interactions. Components are the building blocks of a system, and this level is useful for understanding the system's internal structure.

Finally, the code level provides a view of the actual code that makes up the system. This level is useful for understanding how the components are implemented [37].

The C4 model is an effective way to communicate software architecture, as it provides a clear and consistent way to communicate the different levels of detail and abstraction. It is also a useful tool for designing software systems, as it allows developers and architects to focus on the appropriate level of detail at each stage of the design process.

10.1. SYSTEM CONTEXT LEVEL

At the system context level, there isn't much to show as the proposed architecture can be used in any context, so this level may vary depending on the specific business of the application being built. It's important to note that this level defines the boundaries of the system being built and the external systems or users it interacts with. Therefore, it's crucial to understand the business requirements and the system's stakeholders to determine the appropriate boundaries. At a generic level, this level looks as follows:

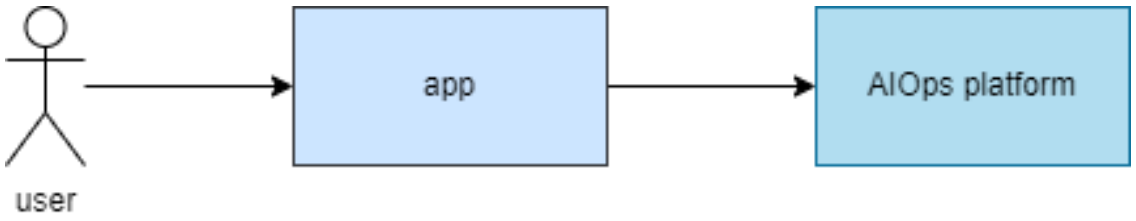

FIGURE 18. SYSTEM CONTEXT VIEW

10.2. CONTAINERS LEVEL

At this level, we cannot provide much detail, however, some applied patterns such as sidecar and asynchronous messaging are already present. We also have a container dedicated to middleware and some monitoring systems.

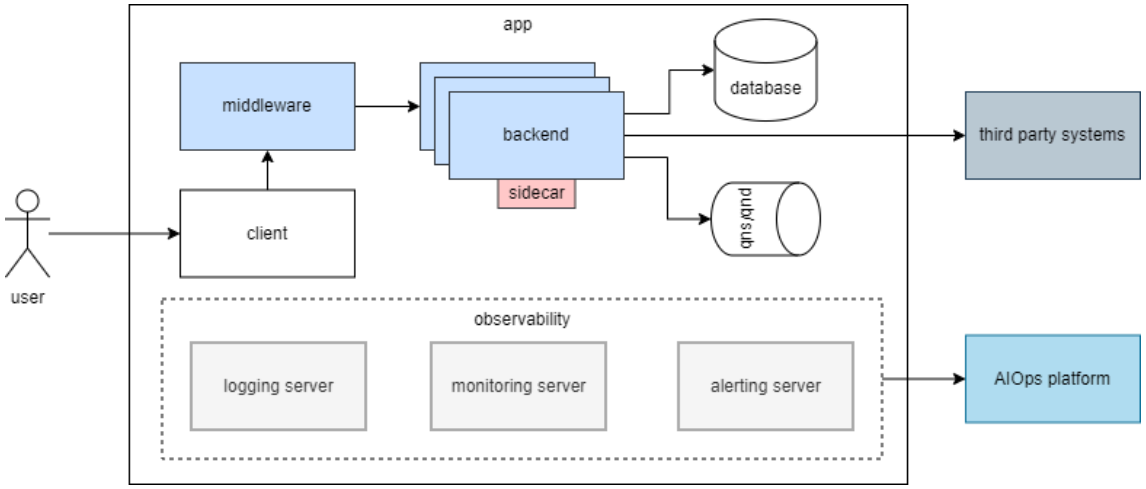

FIGURE 19. CONTAINERS VIEW

It is important to highlight that the logging server implements the log aggregation pattern. Similarly, the monitoring server is used to collect all metrics generated using the application metrics pattern. Finally, the alerting server provides the capabilities to implement the exception tracking pattern. All of this observability section collects and sends data to the AIOps platform. These 3 containers were added separately because they can be handled as standalone systems created from scratch or using existing cloud tools that provide these functionalities.

We also included a container for the client, but this is not considered relevant to the architecture. This can vary depending on the application and may include front-ends, mobile applications, external systems, and more.

The middleware container is critical and added as a layer between the client and the application backend. It can also intercept requests between backend services that communicate with each other. At this point, the API gateway or service mesh pattern can be used depending on the use case, but for ease of implementation, the recommendation is the former for most applications.

In the backend, multiple containers representing the backend are observed, which is typical in distributed systems. Kubernetes is used to manage these containers, which also implements additional patterns such as health check API, replicated load-balanced services, and ownership election. The adapter pattern can also be implemented at this point or in the middleware through

the API gateway. A red container representing the sidecar pattern to be implemented can also be seen. The sidecar pattern can be implemented customly depending on the use case, but Kubernetes provides Envoy Proxy to implement it easily and natively within Kubernetes.

In this diagram, the backend communicates with external systems such as databases, third party systems, and message brokers. While these are optional and may not be present in every use case, they are included since it is very common to connect distributed systems with these types of systems. However, for the architecture being proposed, the main focus will be on the backend itself.

10.3. COMPONENTS LEVEL

It's not possible to provide specific details on which components to use as it will depend on the specific system, but some generic ones are proposed that can be used in most distributed systems applications.

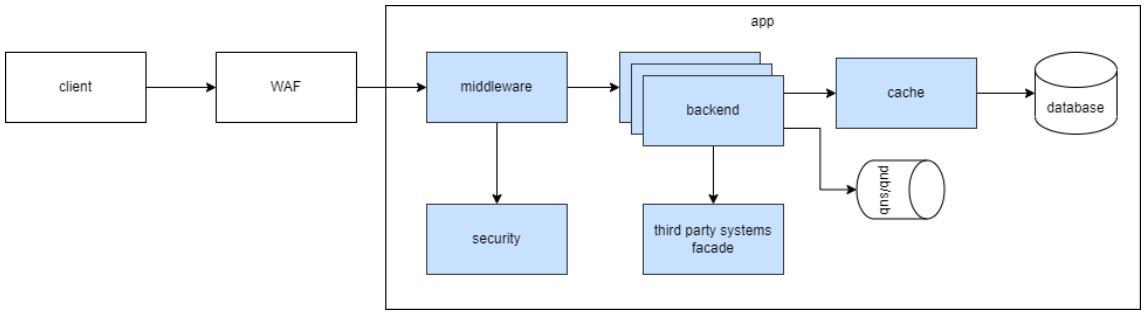

FIGURE 20. COMPONENTS VIEW

In the middleware, we must include security responsibilities. Usually in the API gateway, we can add authentication and authorization concerns, however, it can also communicate with a security component, either internal or external. In addition, an extra layer of security can be added, such as the WAF. This component may be integrated within some API gateways or it may be an external component. Clouds usually provide a ready solution for these security issues.

WAF stands for Web Application Firewall. It is a type of firewall that is specifically designed to protect web applications from various types of attacks, such as cross-site scripting (XSS), SQL injection, and other web-based exploits [39]. WAFs inspect and analyze the HTTP traffic between web applications and clients to identify and block malicious traffic.

A WAF typically works by analyzing incoming requests and filtering out any malicious requests that could exploit web application vulnerabilities [38]. This is done by checking request headers, parameters, and payloads against a set of predefined rules and signatures. If a request matches a known attack signature or violates a security policy, the WAF can either block the request or allow it to pass through after removing the malicious content.

We also have a cache component which is common among distributed systems, since it stores frequently accessed data in memory or a faster storage medium to reduce the response time of read requests. Caching is an effective technique for improving the performance of database-driven applications, as it reduces the number of database queries required to serve

requests [40].

Finally, a component widely used in enterprise and cloud-based systems is the facade for third-party systems. It abstracts communication complexity and provides a simpler and more standardized interface with external third-party systems. It is responsible for handling the communication and translation of data formats and protocols.

10.4. CODE LEVEL

We can not propose a generic code-level view with the C4 model because it is designed to be a high-level model that is independent of any particular programming language or implementation. The model is intended to be a tool for communication and collaboration between stakeholders, allowing them to better understand the system's architecture and make informed decisions. So it depends 100% on the use case. It varies according to the business requirements, the languages, frameworks, and libraries to be used.

The general recommendation is to use code that is easy to maintain and reuse over time, always looking for high cohesion and low coupling between the different components of the application. Try to use clean architectures combined with domain-driven design and a big suite of tests. All of them are easy to implement and improve a lot the code quality. The fewer bugs, the better for the AIOps platform.

11. AIOPS PLATFORM

For the creation of the AIOps platform, the main cloud solutions that were easy to implement and customize were analyzed. The top cloud providers such as AWS, Azure, and GCP provide solutions that are already integrated with artificial intelligence, making it easy to use with just a few configurations. They all provide very similar solutions, so the choice of one or the other is a personal analysis issue. Among all of these, AWS was chosen because it is the one I am most familiar with. Although there are other solutions that offer AIOps as a service, such as Dynatrace, DataDog, Splunk, Moogsoft, PagerDuty, among others, they were not considered due to their high cost of use.

AWS offers a wide range of cloud services that can be easily integrated and scaled based on the needs of the application. It provides a robust set of tools and frameworks for building highly available and fault-tolerant systems, such as Amazon EMR and Amazon Detective, which are essential components for data processing and analysis. Additionally, AWS has strong security and compliance standards, ensuring that the AIOps platform meets industry regulations and protects sensitive data.

Moreover, AWS offers cost-effective pricing models and flexible payment options, which make it a viable option for businesses of all sizes. AWS also has a large community of users and developers, providing a wealth of resources and support for troubleshooting and optimizing the AIOps platform. Lastly, AWS has a strong focus on innovation and continuously introduces new services and features, allowing the AIOps platform to stay up-to-date with the latest advancements in technology.

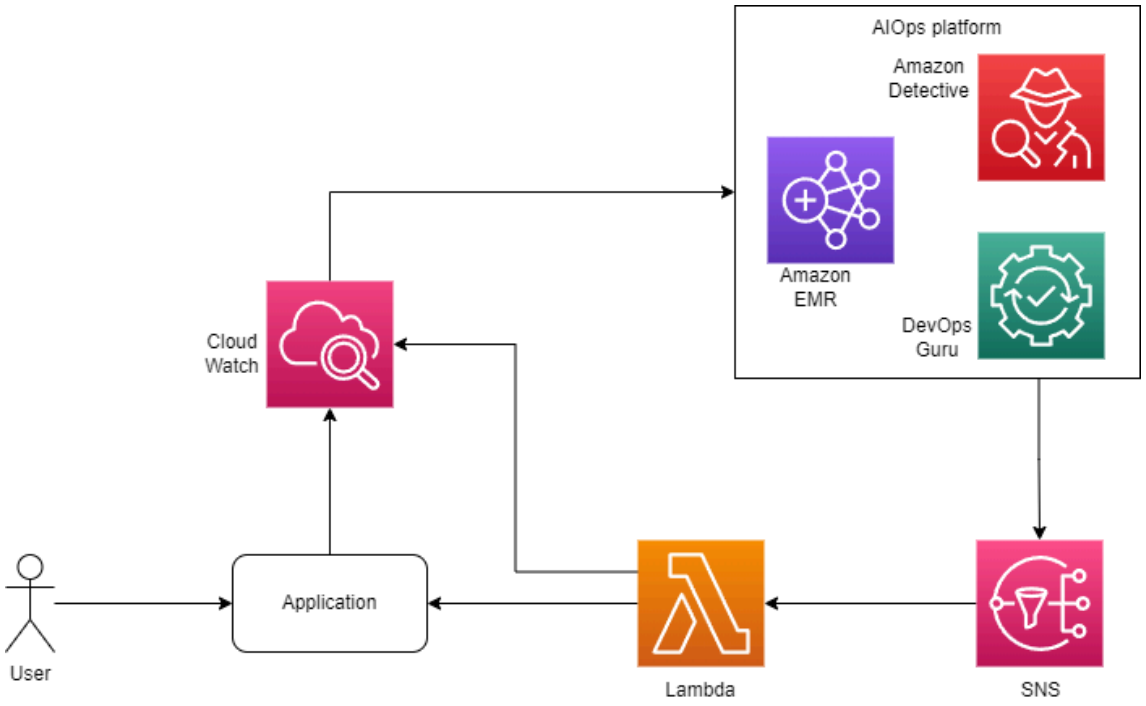

FIGURE 21. AIOPS PLATFORM ARCHITECTURE

In this diagram, we are using the multiple AWS services. One of the core services for most of the applications is Cloud Watch. It allows users to collect and track metrics, collect and monitor log files, and set alarms [41]. CloudWatch provides a unified view of AWS resources, applications, and services that run on AWS, as well as on-premises servers. With CloudWatch, users can monitor and analyze performance metrics of their AWS resources, such as Amazon EC2 instances, Amazon RDS DB instances, and Amazon S3 buckets. Users can also monitor custom metrics generated by their applications and services, and set alarms to notify them of potential issues. CloudWatch also enables users to collect and analyze log data from their applications and systems, making it easier to troubleshoot issues and identify trends. CloudWatch can also be used to troubleshoot performance issues, set up dashboards to visualize metrics, and perform advanced analytics using metric filters and CloudWatch Logs Insights. Additionally, CloudWatch supports integrations with other AWS services, such as Amazon SNS and AWS Lambda, allowing users to take automated actions based on CloudWatch alarms or log data.

We also have AWS Lambda that is a service to run code without the need for managing servers or infrastructure. With Lambda, developers only need to upload their code and set up triggers to execute it. The service automatically provisions and scales the infrastructure needed to run the code in response to the triggers specified by the developer [42].

The AWS SNS is a messaging service that enables the sending and receiving of messages to and from multiple subscribers or endpoints. SNS allows developers to send messages or notifications to a large number of subscribers [43], including mobile devices, email, HTTP/S, and AWS Lambda functions, among others. It is a fully managed service that offers high scalability, availability, and reliability. It allows developers to decouple and separate their systems by sending messages to individual topics. SNS can also be used to trigger other AWS services or Lambda functions, as well as for application-to-application communication, monitoring, and messaging-based architectures.

Now, the core services to create the AIOps platform are Amazon EMR, Amazon Detective, and DevOps Guru. All of them are integrated with artificial intelligence algorithms so we only need to take care of setup the service and they will be able to start working.

Amazon EMR (Elastic MapReduce) is a managed service that enables businesses, researchers, data analysts, and developers to easily and cost-effectively process large amounts of data [44]. It allows users to quickly provision Hadoop clusters and other big data frameworks such as Spark, Hive, HBase, Flink, and Presto, on the AWS Cloud. It provides a simple and scalable way to run big data applications without having to worry about the underlying infrastructure.

Amazon Detective is a service that helps users analyze, investigate, and identify security issues across their AWS resources. It uses machine learning, statistical analysis, and graph theory techniques to quickly analyze trillions of data points from various data sources. By providing a visual representation of the data, Amazon Detective makes it easier for users to quickly identify suspicious activities or potential threats, and drill down to the root cause of the issue [45]. It automatically creates a unified, interactive view of all the relevant data, which can help security teams to reduce the time and effort required to perform manual analysis and investigations.

Amazon DevOps Guru is a machine learning (ML) powered service that helps developers and IT teams improve application reliability and performance [46]. It uses ML algorithms to analyze application telemetry data, such as logs, metrics, and events, to identify operational issues, provide root cause analysis, and make recommendations to resolve or prevent incidents.

The way we send the information to the AIOps platform is:

1. The application generates logs to Cloud Watch.
2. Cloud Watch sends the data to Amazon EMR, Amazon Detective, and DevOps Guru.
3. The Amazon EMR cluster process the logs received from Cloud Watch, generate reports, and send the information again to Cloud Watch.
4. Amazon Detective analyzes the logs from Cloud Watch, generates preventive reports, and sends the information again to Cloud Watch.
5. DevOps Guru receives the data generated in Cloud Watch, including the data from EMR and Detective.
6. DevOps Guru automatically scales the instances in the EKS cluster, before the error happens.
7. SNS topic receives notifications when there is any issue detected in the 3 services and sends the information to Cloud Watch or the EKS cluster using AWS lambda to decide which action applies.

This is a very simple version of AIOps platforms on AWS that will receive and analyze logs generated by your application and provide preventive and corrective actions to keep your application running smoothly.

12. TEST THE AIOPS PLATFORM

In order to test the AIOps platform, we will use a controlled mock scenario to verify the proper communication between the different components within AWS. For this purpose, a simple SpringBoot application was created in Java, exposing a REST endpoint that returns a "Hello World". Additionally, a random error was injected with a 2% probability, to ensure that errors are properly logged and follow the proposed flow accurately.

The Java application was deployed in EKS using a single t2.nano (512 MiB RAM and 1 vCPU) instance for the cluster, as a simple test without auto-scaling was required. To carry out multiple tests on the endpoint, JMeter 5.5 was used, which allows us to simulate loads from multiple users calling the endpoint at the same time.

TABLE 3
AIOPS PLATFORM TEST RESULTS

Number of parallel users	100
Time	30 sec
Number of requests	101476
Number of errors	2537

The created scenario was simple, so multiple requests were made in a very short amount of time and the application didn't have any issues. However, due to the high number of exceptions generated, the instance was affected. As a result of this test, an insight was obtained in the DevOps Guru dashboard regarding the deployed pod in EKS, which was restarted. This demonstrates that the error propagated throughout the AIOps platform flow.

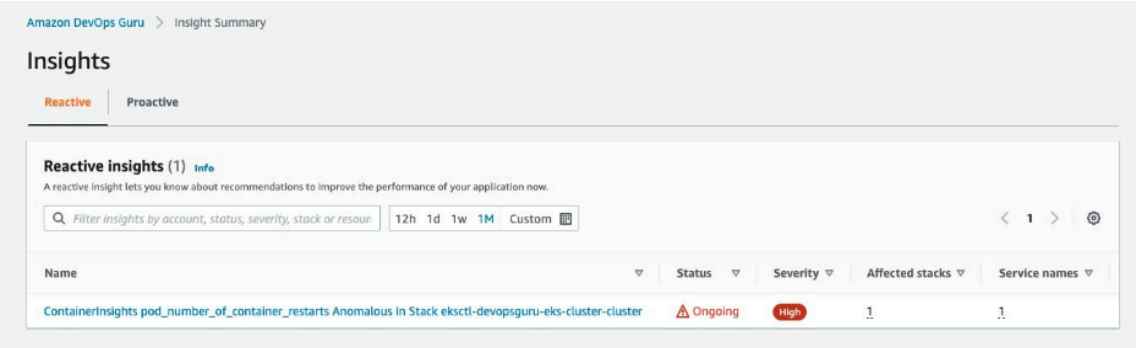

FIGURE 22. DEVOPS GURU INSIGHTS

To improve the accuracy of the AIOps platform's metrics, more realistic use cases need to be created. These use cases should involve multiple parts of a distributed system communicating with each other and performing business operations on HTTP requests.

13. VALIDATION OF THE PROPOSED ARCHITECTURE

To ensure that the proposed architecture integrates well with the AIOps platform, two use cases of distributed systems will be selected. These use cases will use architectures that implement the recommended patterns listed in table 2. Furthermore, a detailed view of the C4 model will be presented, specifically for the Components and Code levels.The other 2 views (system context and containers) are not modeled because in the end they do not change with respect to the one shown in the proposed architecture. This will provide a better understanding of how the different components of the distributed systems interact with each other and how the architecture as a whole is working.

13.1. FIRST USE CASE - MICROSERVICES

The microservices architecture is based on the concept of distributed systems, where an application is divided into smaller, independent components with a single responsibility [49] called microservices. Each microservice runs in its own environment and can communicate with other microservices through communication mechanisms like network calls or messaging. This distribution of functionality into microservices allows for better decoupling and scalability of the application, as each microservice can be developed, deployed, and scaled independently [50].

In a distributed systems approach, microservices can run on different physical or virtual machines, even in different geographic locations [51]. This provides greater resilience and availability since the failure of one microservice does not affect the others. Additionally, the microservices architecture allows for greater flexibility in choosing technologies and programming languages for each microservice, facilitating the adoption of specialized technologies or the upgrade of individual components without impacting the entire application.

Communication between microservices in a distributed microservices architecture can be achieved through various mechanisms such as REST APIs, asynchronous events, or messaging [52]. This enables flexible integration and the ability to scale specific components as needed. However, it also introduces additional challenges such as managing latency, consistency of distributed data, and security in microservice communication.

For this use case, we will model a business that allows users to order food delivery from local restaurants. Although such applications may have multiple functionalities, we will focus on the user's ability to order food for delivery. The component view of the C4 model would look as follows:

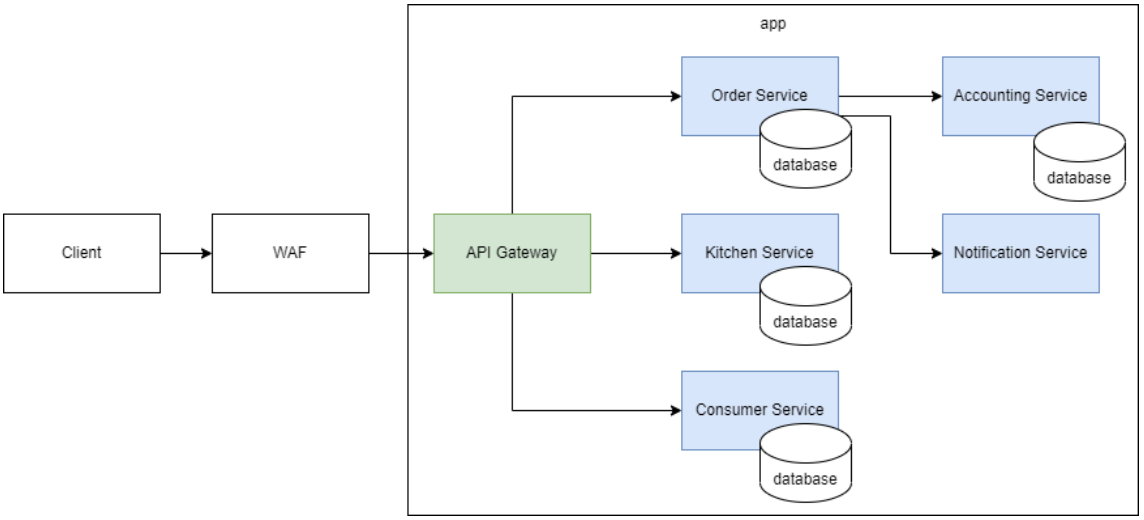

FIGURE 23. COMPONENTS VIEW FOR THE FIRST USE CASE

As we can see, It is a typical microservices architecture where each one has its own database and all of them are behind an API gateway that receives all external HTTP requests and redirects them to the corresponding service.

Each microservice has a unique responsibility. The Order Service is responsible for receiving the order and calling the other microservices to create it. The Kitchen Service validates the order details and checks if they are available for delivery. The Consumer Service manages the information of all customers and verifies that the user can place the order. The Accounting Service authorizes the purchase and verifies the purchase information. Finally, the Notification Service sends messages to the user informing them of updates on the order.

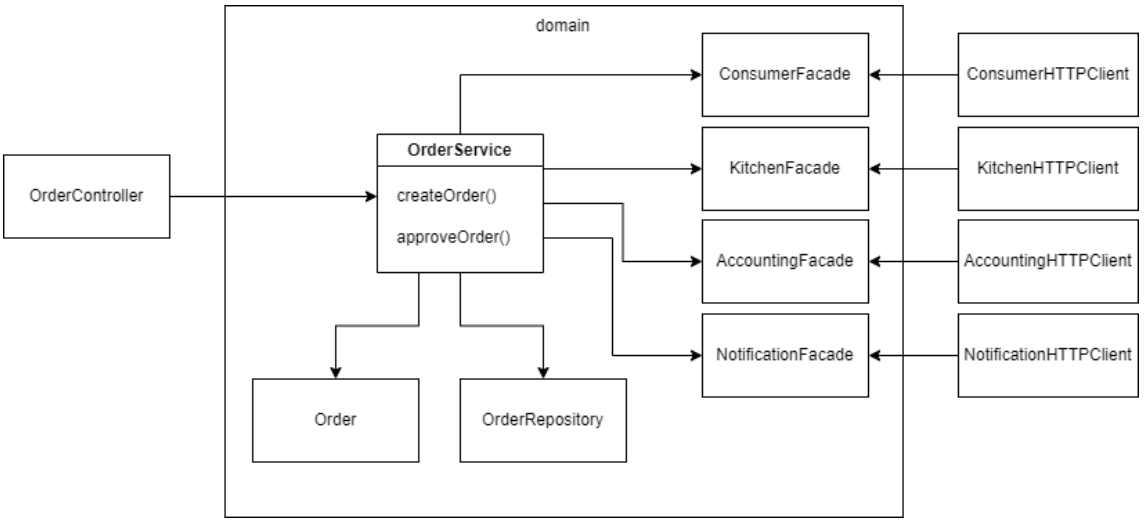

FIGURE 24. CODE VIEW FOR THE ORDER MICROSERVICE

The order microservice is the main one and orchestrates the operations that must be performed to place an order. The user sends an HTTP request, which passes through the controller, which redirects it to the OrderService class, which communicates with the entity and repository to save the information in the database with an APPROVAL-PENDING status. It then queries the Consumer microservice to see if the user requesting the order is authorized to create new orders. Then the Kitchen microservice is called to verify and create the order in the restaurant, and

when a response is obtained, the order status is changed to CREATE-PENDING. Then the Accounting microservice is requested to authorize the payment method, and this changes the status to PAYMENT-CONFIRMED. Finally, the order receives the response of the HTTP requests and changes the status to APPROVED if everything went well, and a last request is launched to the Notification microservice to notify the user.

It is important to note that all requests to external microservices follow a ports and adapters pattern (widely used in clean architecture) with the aim that the domain is independent of any library and only contains business logic. If there is any error in an external call, the order is canceled and the exception is returned to the user.

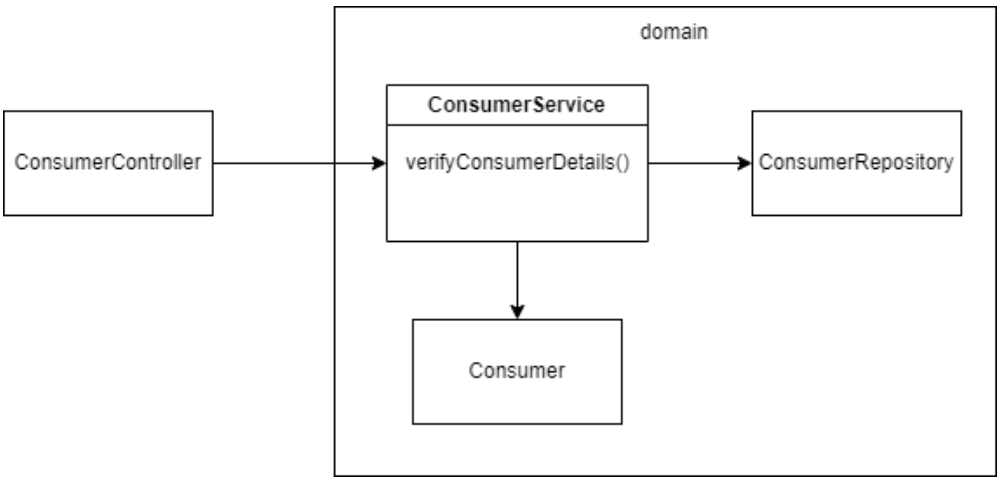

FIGURE 25. CODE VIEW FOR THE CONSUMER MICROSERVICE

The Consumer microservice is quite simple since its only task is to verify the user details and confirm if they are authorized to create more orders. It must check that user is not blocked or have any impediment due to previous orders. In case the verification fails, it will return an exception to the Order service. This information is located in its own database, so the design of this microservice is trivial.

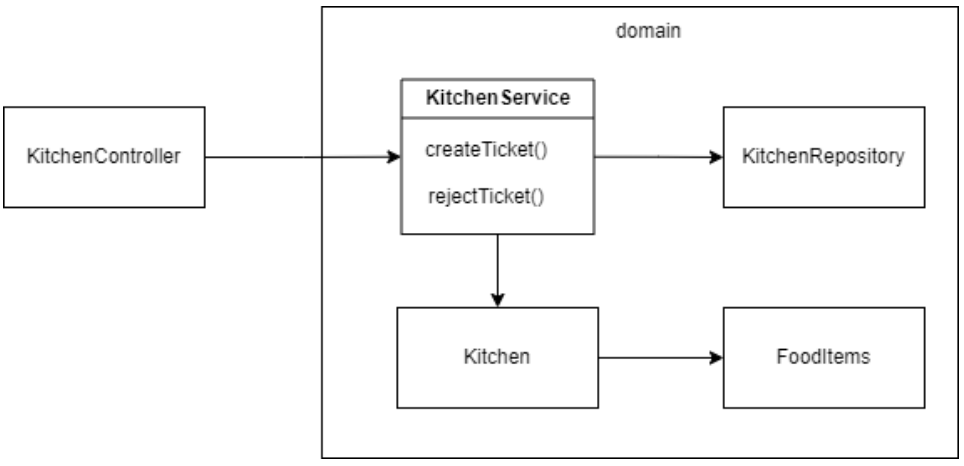

FIGURE 26. CODE VIEW FOR THE KITCHEN MICROSERVICE

The Kitchen microservice is quite similar to the Consumer one in terms of code view, however, in this microservice, there are two operations to create and reject the ticket. In case the

restaurant does not have availability, the order will be rejected. Additionally, the domain entities now consist of an aggregate where the kitchen has many food items.

In some applications of this type, the restaurant's topic requires communication with external systems, however, for this use case, all the information will be available in the database of this Kitchen microservice.

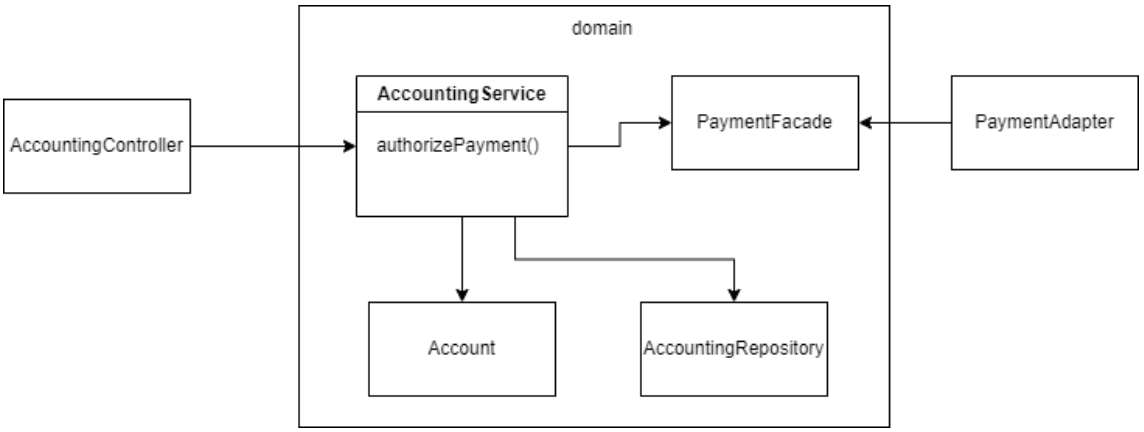

FIGURE 27. CODE VIEW FOR THE ACCOUNTING MICROSERVICE

The Accounting microservice will try to authorize the payment of the order, for which it must connect with a third-party provider that provides that functionality. For this particular example, the PaymentAdapter will simulate a call to an external system but it will always return true. The result of the call is stored in the own database of this service and returned to the Order microservice. In case of any error or if the payment is invalid, an exception is returned.

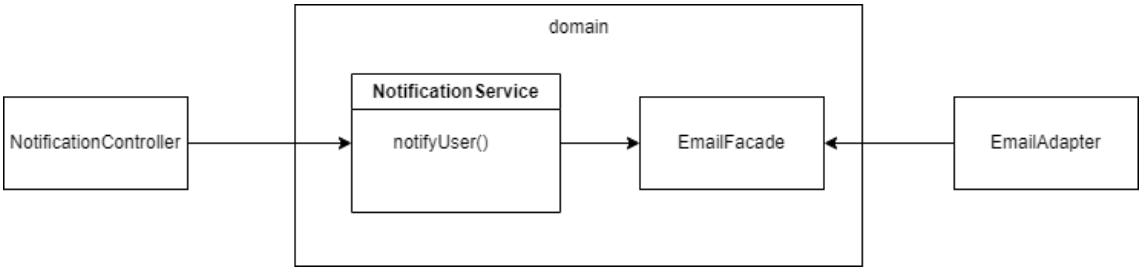

FIGURE 28. CODE VIEW FOR THE NOTIFICATION MICROSERVICE

This microservice will take the notification and send it by email to the user. For this use case, the EmailAdapter will not perform any action since it is not important for the research being conducted. Something curious about this service is that it has no entities or repository since in this case, it is not necessary to save or query anything in the database.

An important point to highlight in all microservices is that they have configuration files that allow them to generate logs of the operations they are performing. This configuration class was introduced in each service using the Microservice Chassis pattern, so for the creation of each microservice, I shared the same base code for them. It also includes the configuration to enable a health check endpoint.

13.2. TEST THE FIRST USE CASE WITH AIOPS PLATFORM

This order food delivery application was deployed in EKS using a t2.nano (512 MiB RAM and 1 vCPU) machine with a single instance for each microservice in the cluster and the opportunity to scale up to 2 instances manually. Load tests were carried out on the order microservice, simulating 100 concurrent users trying to create new orders for 22 minutes. For this test, errors were also injected into all microservices every 5 minutes, so those errors shut down the Java virtual machine, requiring new instances to be created in those cases. DevOps Guru was configured to trigger the alert to EKS and ask it to create a new instance.

TABLE 4
JMETER TEST RESULTS MICROSERVICES

Number of parallel users	100
Time	22 min
Number of requests	1159040
Number of errors	5705

The percentage of failed requests was 0.492%, which is low considering the conditions provided. It is important to note that these results obtained by JMeter reflect the responses of the order microservice, so the behavior of the pods in each microservice should be analyzed separately. These are the results collected from AWS about the health status of the pods:

TABLE 5
MICROSERVICES RESULTS AWS

Microservice	Number of pods created	Average downtime (seconds)	MTTD (seconds)	MTTR (seconds)
Order	5	34.4	24.5	9.9
Consumer	5	21.5	15.2	6.3
Kitchen	5	23.9	15.8	8.1
Accounting	5	24.3	16.1	8.2
Notification	5	20.3	15.2	5.1

It makes sense that all microservices had 5 pods created as they were shut down 4 times plus the initial pod creation. The average downtime is easy to calculate as Cloud Watch provides the exact second when each microservice went down and the exact time when it became available again.

$$average\ downtime\ = \frac{\sum_{0}^{number\ of\ restarts} (time\ first\ successful\ request - time\ first\ failed\ request)}{number\ of\ restarts}$$

For the MTTD, we took the time from when the first failure was recorded in Cloud Watch until the alert was created in DevOps Guru.

$$MTTD = \frac{\sum_{0}^{number\ of\ restarts} (time\ insight\ created - time\ first\ failed\ request)}{number\ of\ restarts}$$

Finally, for the MTTR, we took the moment when the alert was generated and the exact time when the requests started functioning again. All the values were approximated to only one decimal point.

$$MTTR = \frac{\sum_{0}^{number\ of\ restarts} (time\ first\ successful\ request - time\ insight\ created)}{number\ of\ restarts}$$

This experiment resulted in 20 alerts generated in DevOps Guru, corresponding to the 4 times the pods were restarted in each of the 5 microservices. The first alerts were obtained from minute 5:02, which was immediately after the services were shut down, while the last alerts were generated from 19:59, which allowed for scaling even before the microservices failed. This demonstrates the good reaction obtained by the AIOps platform to predict a repetitive error.

Considering the benchmark values obtained in the state of the art for MTTD times of less than 1 minute and MTTR of less than 30 seconds, the results obtained in this case are very good, which demonstrates that the AIOps platform works very well integrated with this use case. It is important to mention that Java was used for these microservices, so the time to create a new instance depends on the application's execution. In this case, the times are still good because each microservice is very lightweight, so they are easy to run.

13.3. ANALYSIS OF THE FIRST USE CASE ARCHITECTURE

Now we need to find out if this architecture truly facilitates integration with AIOps platforms. To do this, we will measure the ease in terms of the non-functional requirements of Adaptability and Interoperability [47]. There is no single way to measure these two requirements, however, a time-based approach is proposed that is easy to validate and contrast.

For adaptability, the following measures were considered:

1. Scalability capability: Scalability can be measured in units of time by the system response time when the workload is increased. For example, the time required to process a request when the number of users is increased. In this case, when a single request is launched, the average response time is 511 milliseconds, when 100 concurrent users are launched the average time is 726 milliseconds, and if 1000 users are launched in parallel the average time is 759 milliseconds. It can be evidenced that the application is not degrading significantly, so it scales correctly.
2. Robustness of a system: It measures a system's ability to maintain its operation despite errors or changes in its environment. This metric can be quantified by the frequency and severity of errors and the system's ability to recover from them. Thanks to the results obtained in AWS, we can conclude that the application recovers in less than 10 seconds, demonstrating its robustness.

3. Flexibility: Flexibility can be measured in units of time by the time required to make a change in the system. For example, the time required to add new functionality or to adapt the system to a new environment. This will be measured later.

For interoperability, the following measures were considered:

1. Response time: The response time is the period of time it takes for a system to respond to a request from another system. This time can be a good metric for measuring interoperability in terms of time, especially in real-time applications. In the test performed, the average response time was 738 milliseconds, which aligns with the number of operations to be performed.
2. Integration time: Integration time is the period of time it takes to integrate two or more systems to work together. This time can be a good metric to measure interoperability in terms of time. This will be measured later.

Flexibility and integration time are two metrics that require input from multiple individuals to avoid biases in the results. For this exercise, the evaluation techniques of Experiments and Surveys, as explained in the book *Experimentation in Software Engineering* [54] were combined. This approach involves conducting controlled experiments and then using surveys to obtain the participants perceptions. It is a great option for measuring complex topics that are subjective and do not have a single, exact way of measurement. It is ideal for measuring these two non-functional requirements. In this way, a more comprehensive understanding of the architecture's impact on adaptability and interoperability can be achieved.

An analysis was conducted on a sample of 22 Java developers from Colombia to add integration of the Consumer microservice from scratch with the AIOps platform through the logs to be generated. The code for the microservice was provided to them without the chassis that included the configuration to send the logs to Cloud Watch. With this experiment, we can measure the integration time. After integrating it, they were asked to modify the configuration to also generate logs in a local plain file. This way, the flexibility of the architecture to modify that functionality was measured. Finally, they were asked to fill out a survey where they were asked about the time spent on each of the two tasks in order to obtain a real average time.

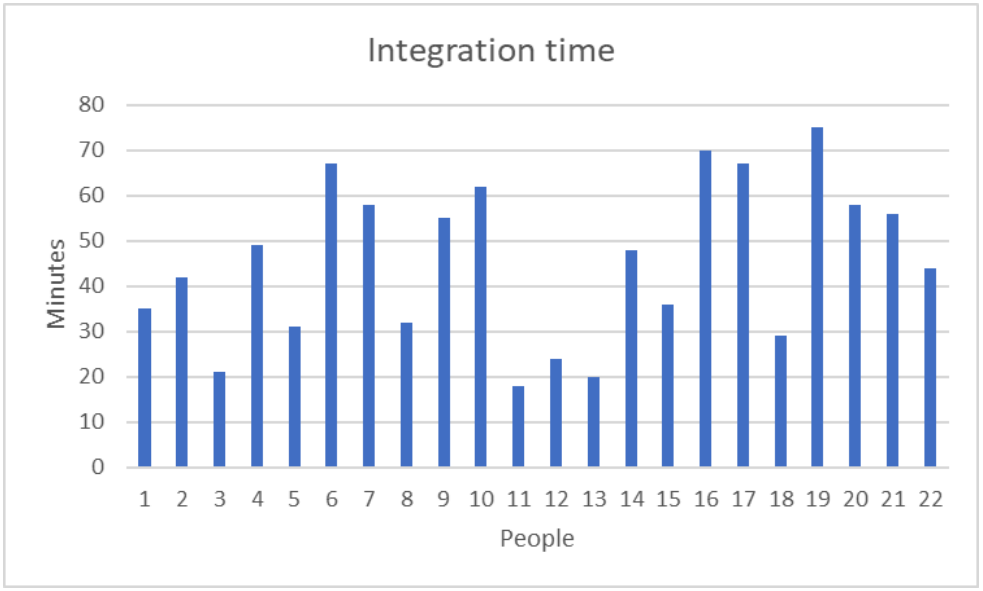

FIGURE 29. INTEGRATION TIME MICROSERVICES

The average integration time spent was 44.8 minutes. This means that for a person adopting the architecture and wishing to integrate with an AIOps platform, the time investment required is less than an hour, which is considered a very good result.

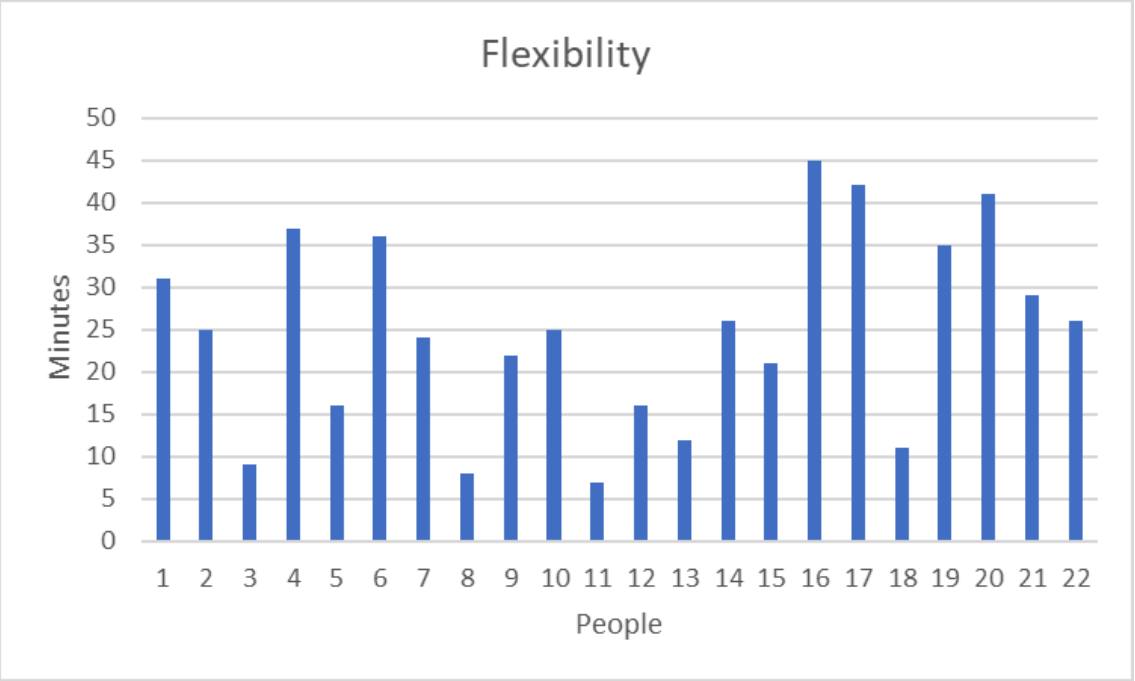

FIGURE 30. FLEXIBILITY MICROSERVICES

The average time spent on flexibility was 24.7 minutes. This means that for someone using the architecture, it takes less than half an hour to make modifications to it to add new functionality, which is considered a very positive result.

Given the results of the 5 metrics obtained, we can conclude that the architecture implemented in this use case using microservices is adaptable and interoperable. Therefore, we can say that it facilitates integration with AIOps platforms.

13.4. SECOND USE CASE - SERVERLESS

The serverless architecture or Function-as-a-Service (FaaS), built on the foundation of distributed systems, represents an evolution of microservices. While microservices focus on breaking down monolithic applications into smaller, independent services, serverless takes this concept further by eliminating the need for infrastructure provisioning and management. In a serverless architecture, services are implemented as functions that are executed in a distributed manner across multiple nodes or containers.

By leveraging distributed systems, serverless architectures can achieve greater scalability, fault tolerance, and resource efficiency. Functions can be executed in parallel across different nodes, allowing for elastic scaling based on demand [53]. The distributed nature of the architecture also enables fault tolerance, as functions can be automatically replicated and distributed across

multiple nodes to ensure high availability.

Moreover, the serverless model simplifies deployment and operations by abstracting away the underlying infrastructure. Developers can focus solely on writing code for individual functions without worrying about managing servers or containers. This shift in responsibility allows for faster development cycles, increased agility, and reduced operational overhead.

Lets take the same business requirement of the previous use case to allow users to order food delivery from local restaurants focused on order creation. The component view of the C4 model would look as follows:

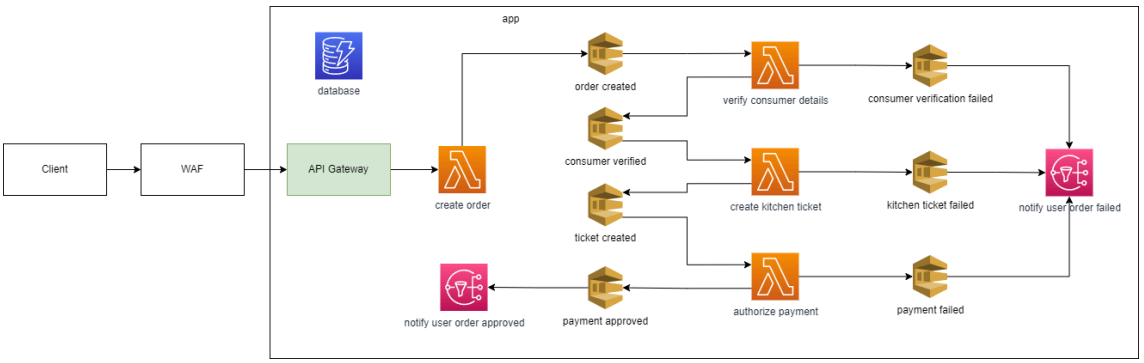

FIGURE 31. COMPONENTS VIEW FOR THE SECOND USE CASE

In this case, we have the same functions that were previously part of each microservice, but now they are deployed as lambda functions within AWS. An important change from the previous case is that communication now occurs using the asynchronous messaging pattern (as we saw in the list of recommended patterns). Each function generates a result, which is then published to a queue that the next component, which must be executed, is subscribed to. In the case of any errors, it is published to an error queue, and an error notification is sent.

Another important change is that we no longer have a notification service, but instead, we use the SNS service directly, which allows for easy notification sending. Additionally, information is stored in a Dynamo database, which is easy to configure for lambda functions and is also 100% serverless. Each lambda function performs a process on the HTTP request that comes from the API Gateway and saves the information in the database. The following functions also store information and modify the order state, as we saw in the previous use case.

In this case, we are not deploying on Kubernetes, as we saw in the proposed architecture since lambda functions are virtualized components that are fully managed by AWS. It still has the same benefits and uses the same patterns that we saw in the proposed architecture. However, it will be much more managed now.

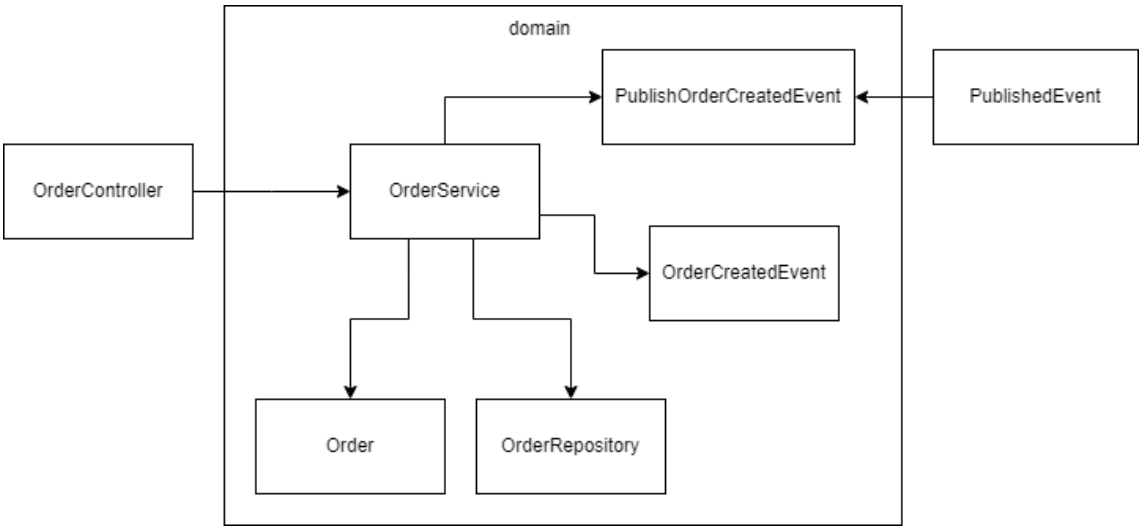

FIGURE 32. CODE VIEW FOR CREATING ORDER FUNCTION

Similar to what we saw in the microservices case, but now much more simplified since this function only takes care of creating the order in the database and publishing the event that it was created. Also, clean architecture is still followed in the case of publishing events, as the implementation remains in the infrastructure layer and the domain only depends on an interface.

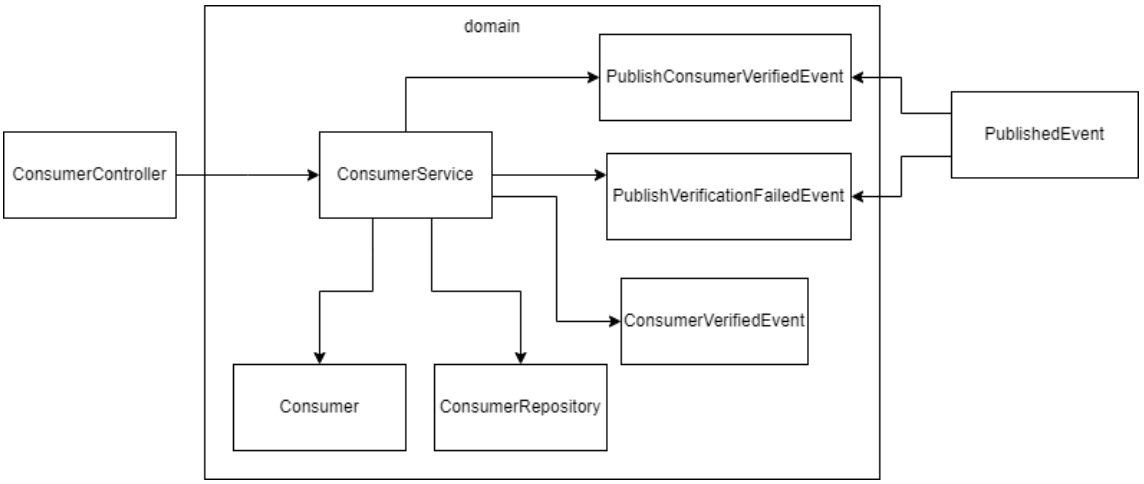

FIGURE 33. CODE VIEW FOR VERIFYING CONSUMER FUNCTION

In this case, two events can be published for both successful and failed cases. Both interfaces are implemented in the infrastructure layer, and the domain layer decides which of the two events to invoke.

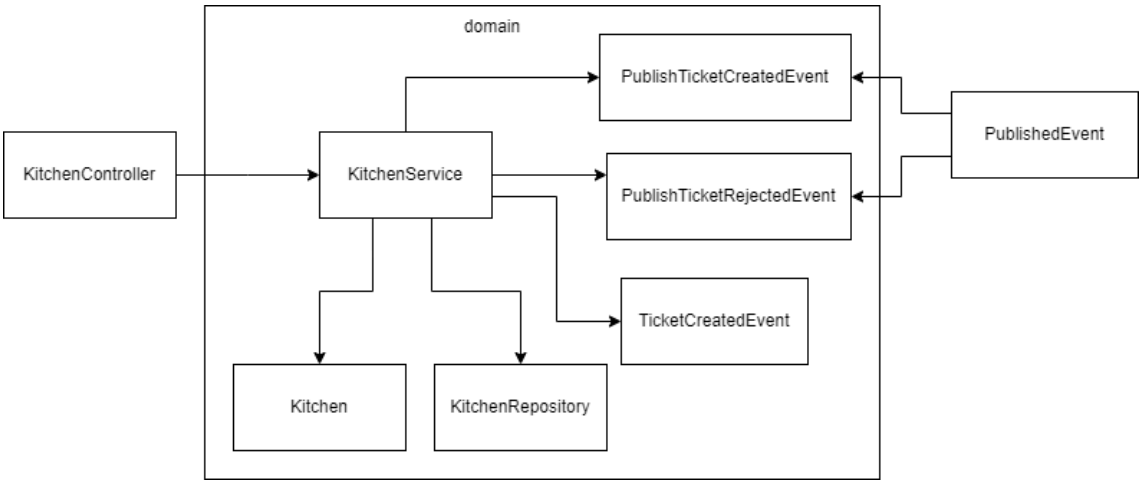

FIGURE 34. CODE VIEW FOR CREATING KITCHEN TICKET FUNCTION

The same structure of the function as for Consumer, but now for Kitchen. Also, we have 2 events to publish for the successful and failed cases.

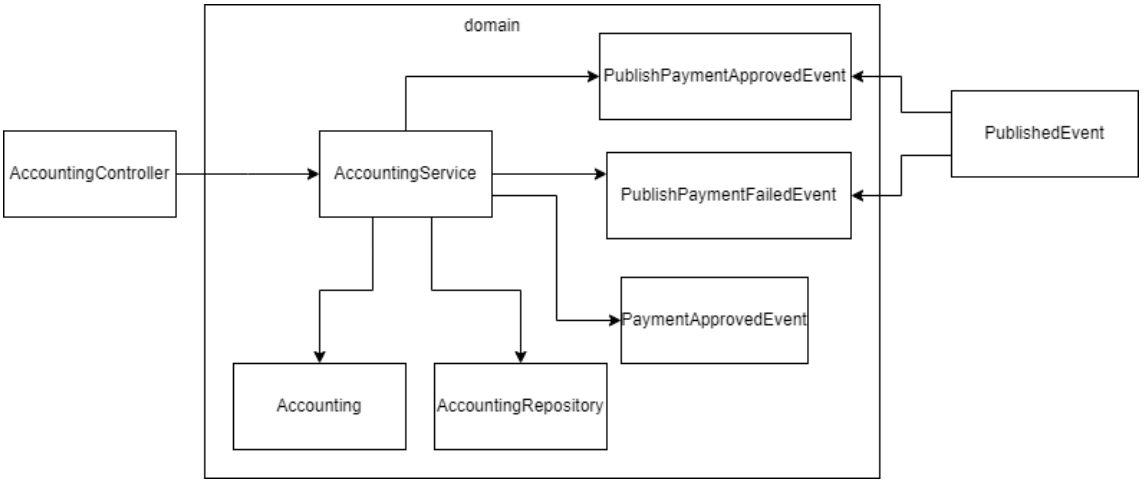

FIGURE 35. CODE VIEW FOR PAYMENT AUTHORIZATION FUNCTION

The same structure as the Consumer and Kitchen functions, following the same patterns to create the classes.

It is important to highlight that just like in the use case with microservices, the microservice chassis pattern was implemented to create functions with a predefined structure. It contains the necessary configuration classes to handle connections with AWS systems such as Cloud Watch, SQS, and DynamoDB. In this case, the health check endpoint was not configured, as by default lambda functions can detect when they fail automatically.

13.5. TEST THE SECOND USE CASE WITH THE AIOPS PLATFORM

This case is special because the same architectural pattern, such as serverless, indicates how each component of the system should be deployed. Each function is deployed on a lambda, which is a service managed by AWS that handles 100% of the function's lifecycle based on the

received request [48]. Java 17 was used as the programming language, and, as in the case of microservices, the API Gateway, which is managed by AWS, is used. The simulation of 100 concurrent users trying to create new orders for 22 minutes was also performed. For this test, errors were also injected into all the lambda functions every 5 minutes, so those errors shuts down the Java virtual machine, requiring new instances to be created in those cases. DevOps Guru was configured to generate insights.

TABLE 6
JMETER TEST RESULTS SERVERLESS

Number of parallel users	100
Time	22 min
Number of requests	2842191
Number of errors	1357

The percentage of failed requests was 0.0477%, which is lower than the microservice use case. It is important to note that these results obtained by JMeter reflect the responses of the create order function, so the behavior of the pods in each function should be analyzed separately. These are the results collected from AWS about the health status of the functions:

TABLE 7
SERVERLESS RESULTS AWS

Function	MTTD (seconds)	MTTR (seconds)
Create order	2.1	4.5
Verify consumer details	1.5	2.0
Create kitchen ticket	1.8	1.5
Authorize payment	2.2	1.7

In this case, the average downtime is not calculated because each lambda function runs on independent Linux containers that can be reused for some requests. Running independently, an error in one request does not affect other requests, so there is no specific moment when the application is down. Only one instance of the multiple instances created by AWS falls down and it is recreated on demand afterward. However, MTTD was identified by measuring the time it took from when the log was registered in CloudWatch until the alert was created in DevOps Guru. For MTTR, the time was taken from when the requests failed until the next request was responded to successfully. In this case, the recovery time is extremely short due to the inherent nature of serverless for managing concurrency.

107 alerts were registered within DevOps Guru, which served only to report the errors presented since the scalability was directly handled by AWS Lambda functions. At the time of the errors, more than 20 Lambda functions were running simultaneously, so the number of alerts increased compared to the microservices case. However, these alerts can be used preventively in larger applications, where infrastructure administrators can proactively receive the insights generated

by this tool and take actions in advance.

In this case, there is a significant improvement in the times compared to the reference times obtained, with detection times of less than 3 seconds and recovery times of less than 5 seconds. Like in the first use case, Java was used as the programming language, using Spring Boot to handle the HTTP requests.

13.6. ANALYSIS OF SECOND USE CASE SERVERLESS

Similar to the microservices use case, we will measure the proposed architecture in terms of Adaptability and Interoperability. Using the same 5 characteristics, we will quantify with numbers the impact that the design of this architecture has, at least at the level of ease.

For adaptability:

1. Scalability capability: Due to the auto-scalability provided by the serverless architectures, and the fact that each request can be executed in an isolated instance, this greatly facilitates the scalability of the solution, since no matter how many requests are made to the endpoint, it will always be able to respond without degrading response times. When requests were made with a single user, an average response time of 553 milliseconds was obtained, with 100 users it was 549 milliseconds, and with 1000 users it was 561 milliseconds. The application scales properly.
2. Robustness of a system: The system recovered automatically and easily from the errors presented in the controlled environment, with recovery times of less than 5 seconds, which is incredibly fast considering that it runs a Java application for each request. The system is very robust.
3. Flexibility: Similarly to the previous use case, a population sample of developers were taken, who were asked to make modifications to the application, and at the end, the reported times were taken into account to draw a conclusion. In this experiment, 14 Java developers from Colombia participated and were asked to send logs to CloudWatch before publishing messages to SQS with the body of the event.

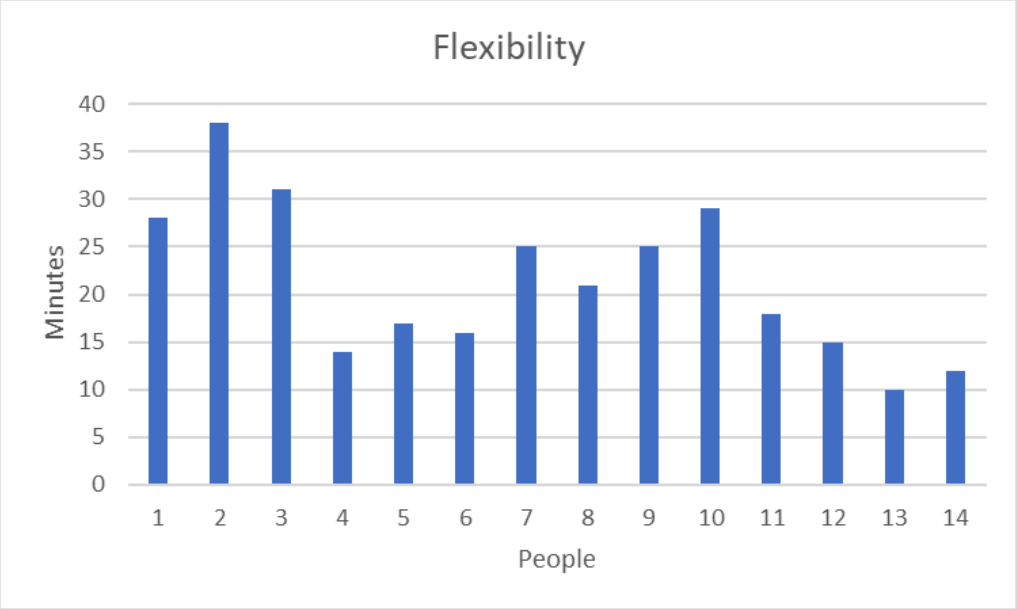

FIGURE 36. FLEXIBILITY SERVERLESS

The average time spent on applying these changes was 21.35 minutes. This means that for someone using the architecture, it takes a few minutes to make modifications to it to add new functionality, which is considered a very positive result.

For interoperability, the following measures were considered:

1. Response time: As in the microservices case, this metric is easy to measure thanks to the tests carried out with JMeter on the application. The average response time was 564 milliseconds.
2. Integration time: The 14 developers were asked to measure how long it takes them to integrate with the AIOps platform so that logs are sent at the beginning of each request in the controller classes. For this, they were given the code of a lambda function and asked to add the necessary configurations and code to send logs to CloudWatch.

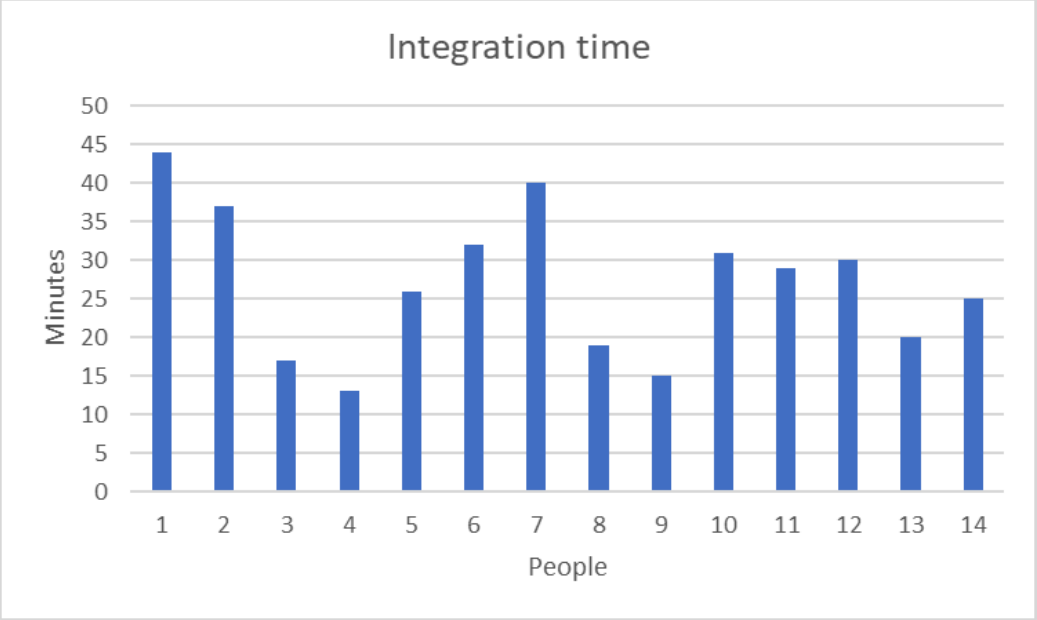

FIGURE 37. INTEGRATION TIME SERVERLESS

The average integration time spent was 27 minutes. This means that for a person adopting the architecture and trying to integrate with an AIOps platform, the time investment required is less than half an hour, which is considered a very good result.

Given the results of the 5 metrics obtained, we can conclude that the architecture implemented in this use case using serverless is adaptable and interoperable. Therefore, we can say that it facilitates integration with AIOps platforms.

14. CONCLUSIONS

Out of the 16 patterns analyzed, 11 were selected, 7 of which correspond to monitoring patterns. These were used to build the proposed architecture named AIODS, in which the use of monitoring patterns, middlewares, and virtualization was prioritized as analyzed in the state of the art. The recommendations provided at each level of the C4 model can be observed, however, for the component and code levels, a greater context of the use case where this architecture will be applied is required.

Using cloud-native solutions facilitates the implementation of AIOps platforms, especially in clouds like AWS where there is a great advancement in the integration of artificial intelligence with the services offered. It can be evidenced how simple it is to connect different services which, as a result, provide valuable insights that allow predicting problems in distributed systems. The more self-managed a solution is, the fewer responsibilities for programmers.

Although the validation with the 2 use cases required extensive analysis, the benefits provided by the proposed architecture were successfully evaluated. In both use cases, a a low error rate of less than 1% was obtained, and error detection and correction times were below the standards collected in the state-of-the-art analysis (MTTD of 1 minute and MTTR of 30 seconds). In both use cases, the architecture used was varied but always taking into account what was proposed, seeking to integrate the proposed patterns.

As a result of each use case, the extent to which the architecture facilitates integration with AIOps platforms was evaluated, taking into account two non-functional requirements: adaptability and interoperability. Metrics such as scalability capability, robustness of the system, and flexibility were used to measure adaptability, while response time and integration time were used to measure interoperability. Evaluating a sample population was necessary to avoid biases in some of these measures. Finally, consistent results were obtained that demonstrate the ease of integration with AIOps platforms provided by the proposed architecture.

The proposed architecture provides a clear blueprint for designing, deploying, and managing distributed systems in a way that aligns with AIOps principles and it is easy to adapt for any business problem. It ensures that the various components, such as middlewares, monitoring systems, and virtualization, are properly integrated to support AIOps capabilities effectively. By following this architecture, organizations can ensure easier integration between distributed systems and AIOps platforms. This integration enables the collection, analysis, and interpretation of vast amounts of operational data in real-time, leading to actionable insights, automated incident detection, and proactive problem resolution.

Companies that want to adopt the relatively new concept of AIOps will be able to integrate it much more easily, which will remove the barrier for more applications to start taking advantage of the benefits provided by AIOps platforms. As a result, we will have applications with less downtime. It also provides operational efficiency, increased availability and performance, resource optimization and informed decision making. By making more accurate and timely strategic and tactical decisions, organizations can minimize risk, maximize opportunity, and gain competitive advantage.

15. FUTURE WORK

As time goes on, there will be many more and better options for AIOps platforms. The revolution of artificial intelligence, significant advancements in machine learning and big data will enable solutions with greater precision in predicting failures, making them increasingly autonomous.

In addition to the general implementation proposed in the article, an interesting approach would be to develop domain-specific and domain-agnostic implementations for different sectors or industries. This would involve adapting the AIOps architecture and tools used to meet the specific requirements and challenges of each domain.

For example, in the financial sector, specific data sources such as market feeds, transaction data, and key financial metrics could be integrated to enhance the detection and analysis of performance and security anomalies in financial systems. Similarly, in the healthcare sector, electronic medical records, connected medical devices, and patient monitoring data could be incorporated to improve early detection of critical issues and enhance the quality of care.

Furthermore, to ensure portability and choice of cloud service providers, it would be valuable to conduct multi-cloud testing to compare and evaluate the offerings from major cloud service providers such as AWS, GCP, and Azure. This would help identify differences in AIOps capabilities, integration with other tools and services, scalability, and performance across different cloud environments. These multi-cloud tests would enable organizations to make informed decisions regarding the selection of the most suitable cloud service provider for their needs and leverage the full potential of AIOps capabilities in that specific environment.

Additionally, it would be interesting to explore and analyze other architectural patterns that can be applied to different areas of the distributed system. While the article has focused on 3 key sections, there are numerous other patterns worth investigating. For example, patterns related to data management and real-time processing can offer valuable insights and solutions to specific challenges within the architecture.

As more use cases and requirements arise, it becomes crucial to continuously evaluate the architecture performance and effectiveness. This involves considering more non-functional requirements focused in each use case to analyze. It will provides opportunities to identify areas of improvement and make informed decisions on architectural adjustments or enhancements.

REFERENCES

[1] S. Shen, J. Zhang, D. Huang and J. Xiao, "Evolving from Traditional Systems to AIOps: Design, Implementation and Measurements," 2020 IEEE International Conference on Advances in Electrical Engineering and Computer Applications(AEECA), 2020, pp. 276-280, doi: 10.1109/AEECA49918.2020.9213650.

[2] A. Levin et al., "An Anomaly Detection Algorithm for Microservice Architecture Based on Robust Principal Component Analysis" 2019 IEEE International Congress on Big Data (BigDataCongress), 2019, pp. 165-169, doi: 10.1109/BigDataCongress.2019.00036.

[3] Yangguang Li, Zhen Ming (Jack) Jiang, Heng Li, Ahmed E. Hassan, Cheng He, Ruirui Huang, Zhengda Zeng, Mian Wang, and Pinan Chen. 2020. A Context Model for Holistic Monitoring and Management of Complex IT Environments. ACM Trans. Softw. Eng. Methodol. 29, 2, Article 13 (April 2020), 24 pages. DOI:https://doi.org/10.1145/3385187

[4] S. Nedelkoski, J. Cardoso and O. Kao, "AI-Governance and Levels of Automation for AIOps-supported System Administration" 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2019, pp. 241-250, doi: 10.1109/CCGRID.2019.00038.

[5] Notaro P., Cardoso J., Gerndt M. (2021) Ananke: A framework for Cloud-Native Applications smart orchestration. In: Hacid H. et al. (eds) Service-Oriented Computing – ICSOC 2020 Workshops. ICSOC 2020. Lecture Notes in Computer Science, vol 12632. Springer, Cham. https://doi.org/10.1007/978-3-030-76352-7_15

[6] J. Cardoso and O. Kao, "Managing Distributed Cloud Applications and Infrastructure A Self-Optimising Approach. Chapter 3: Application Optimisation: Workload Prediction and Autonomous Autoscaling of Distributed Cloud Applications" 2020, pp. 80-90.

[9] Notaro, P.; Cardoso, J. and Gerndt, M, "A Systematic Mapping Study in AIOps", International Workshop on Artificial Intelligence for IT Operations (AIOPS) 2020.

[10] Kobbacy, K.A.H., Vadera, S., Rasmy, M.H.: AI and OR in management of operations: history and trends. Journal of the Operational Research Society 58(1), 10–28 (Jan 2007). https://doi.org/10.1057/palgrave.jors.2602132

[11] Mukwevho, M.A., Celik, T.: Toward a Smart Cloud: A Review of Fault-tolerance Methods in Cloud Systems. IEEE Transactions on Services Computing pp. 1–1 (2018). https://doi.org/10.1109/tsc.2018.2816644

[12] L. Li, R. Hansman, R. Palacios and R. Welsch, "Anomaly Detection via a Gaussian Mixture Model for Flight Operation and Safety Monitoring", Transportation Research Part C-Emerging Technologies, vol. 64, pp. 45-57, 2016.

[13] M. Farshchi, J. Schneider, I. Weber and J. Grundy, "Metric Selection and Anomaly Detection for Cloud Operations Using Log and Metric Correlation Analysis", Journal of Systems and Software, vol. 137, pp. 531-549, 2017.

[14] S. Nedelkoski, J. Cardoso and O. Kao, "Anomaly Detection from System Tracing Data Using Multimodal Deep Learning", 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 179-186, 2019.

[15] M. Du, S. Versteeg, J. Schneider, J. Han and J. Grundy, "Interaction Traces Mining for Efficient System Responses Generation", ACM Sigsoft Software Engineering Notes, vol. 40, no. 1, pp. 1-8, 2015.

[16] F. Salfner, M. Lenk and M. Malek, "A Survey of Online Failure Prediction Methods", ACM Computing Surveys, vol. 42, no. 3, pp. 10, 2010.

[17] Z. Wang, M. Zhang, D. Wang, C. Song, M. Liu et al., "Failure Prediction Using Machine Learning and Time Series in Optical Network", Optics Express, vol. 25, no. 16, pp. 18553-18565, 2017.

[18] H. Wang and H. Zhang, "AIOPS Prediction for Hard Drive Failures Based on Stacking Ensemble Model", 2020 10th Annual Computing and Communication Workshop and Conference (CCWC), pp. 417-423, 2020.

[19] Q. Chen, Z. Zheng, C. Hu, D. Wang and F. Liu, "Data-Driven Task Allocation for Multi-Task Transfer Learning on the Edge", 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 1040-1050, 2019.

[20] Y. Dang, Q. Lin and P. Huang, "AIOps: Real-World Challenges and Research Innovations," 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2019, pp. 4-5, doi: 10.1109/ICSE-Companion.2019.00023.

[21] Burns, B, "Designing distributed systems: patterns and paradigms for scalable, reliable services", 2018, O'Reilly Media, Inc.

[22] Fan, C. F., Jindal, A., & Gerndt, M, "Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application", 2020,. In CLOSER (pp. 204-215).

[23] Richardson, C, "Microservices Patterns", 2019, Manning Publications.

[24] Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural patterns for microservices: a systematic mapping study. In CLOSER

2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018. SciTePress.

[25] Boulon, J., Konwinski, A., Qi, R., Rabkin, A., Yang, E., & Yang, M. (2008, October). Chukwa, a large-scale monitoring system. In Proceedings of CCA (Vol. 8, pp. 1-5).

[26] Shkuro, Y. (2019). Mastering Distributed Tracing: Analyzing performance in microservices and complex systems. Packt Publishing Ltd.

[27] Cassé, C., Berthou, P., Owezarski, P., & Josset, S. (2022, January). A tracing based model to identify bottlenecks in physically distributed applications. In 2022 International Conference on Information Networking (ICOIN) (pp. 226-231). IEEE.

[28] Waseem, M., Liang, P., Shahin, M., Di Salle, A., & Márquez, G. (2021). Design, monitoring, and testing of microservices systems: The practitioners' perspective. Journal of Systems and Software, 182, 111061.

[29] Pai, V. S., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., & Nahum, E. (1998). Locality-aware request distribution in cluster-based network servers. ACM SIGOPS Operating Systems Review, 32(5), 205-216.

[30] Dahl, O. (2002). Enterprise application integration. School of Mathematics and Systems Engineering.

[31] Moguel, E., Rojo, J., Valencia, D., Berrocal, J., Garcia-Alonso, J., & Murillo, J. M. (2022). Quantum service-oriented computing: current landscape and challenges. Software Quality Journal, 30(4), 983-1002.

[32] Gadge, S., & Kotwani, V. (2018). Microservice architecture: API gateway considerations. GlobalLogic Organisations, Aug-2017, 11.

[33] Li, W., Lemieux, Y., Gao, J., Zhao, Z., & Han, Y. (2019, April). Service mesh: Challenges, state of the art, and future research opportunities. In 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE) (pp. 122-1225). IEEE.

[34] Nusairat, J. F., & Nusairat, J. F. (2020). Deployment. Rust for the IoT: Building Internet of Things Apps with Rust and Raspberry Pi, 289-389.

[35] Shah, J., & Dubaria, D. (2019, January). Building modern clouds: using docker, kubernetes & Google cloud platform. In 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC) (pp. 0184-0189). IEEE.

[36] A. Vázquez-Ingelmo, A. García-Holgado and F. J. García-Peñalvo, "C4 model in a Software Engineering subject to ease the comprehension of UML and the software," 2020 IEEE Global Engineering Education Conference (EDUCON), Porto, Portugal, 2020, pp. 919-924, doi: 10.1109/EDUCON45650.2020.9125335.

[37] V. Stirbu and T. Mikkonen, "CompliancePal: A Tool for Supporting Practical Agile and Regulatory-Compliant Development of Medical Software," 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), Salvador, Brazil, 2020, pp. 151-158, doi: 10.1109/ICSA-C50368.2020.00035.

[38] Clincy, V., & Shahriar, H. (2018, July). Web application firewall: Network security models and configuration. In 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC) (Vol. 1, pp. 835-836). IEEE.

[39] A. Razzaq, A. Hur, S. Shahbaz, M. Masood and H. F. Ahmad, "Critical analysis on web application firewall solutions," 2013 IEEE Eleventh International Symposium on Autonomous Decentralized Systems (ISADS), Mexico City, Mexico, 2013, pp. 1-6, doi: 10.1109/ISADS.2013.6513431.

[40] Noah, P., & Seman, S. (2012). Distributed multi-level query cache: the impact on data warehousing. Issues in Information Systems, 13(2), 51-57.

[41] Guide, D. (2009). Amazon CloudWatch.

[42] Sbarski, P., & Kroonenburg, S. (2017). Serverless architectures on AWS: with examples using Aws Lambda. Simon and Schuster.

[43] Srinivasan, A., Natarajan, N., Karunakaran, R. V., Elangovan, R., Shankar, A., Sabharish, P. M., ... & Radha, S. (2020, December). Elder care system using iot and machine learning in aws cloud. In 2020 IEEE 17th International Conference on Smart Communities: Improving Quality of Life Using ICT, IoT and AI (HONET) (pp. 92-98). IEEE.

[44] JS, S. M. CDPSM: A New Optimized Progressive Big Data Analytics For Partial Cancer Data using Amazon EMR.

[45] Badhwar, R. (2021). Cloud Monitoring Security Controls for AWS. In The CISO's Next Frontier: AI, Post-Quantum Cryptography and Advanced Security Paradigms (pp. 297-307). Cham: Springer International Publishing.

[46] N. Sawant and S. H. Sengamedu, "Learning-based Identification of Coding Best Practices from Software Documentation," 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), Limassol, Cyprus, 2022, pp. 533-542, doi: 10.1109/ICSME55016.2022.00073.

[47] Paz, J. A. M., Gómez, M. Y. M., & Rosas, S. C. (2017, July). Análisis sistemático de información de la Norma ISO 25010 como base para la implementación en un laboratorio de Testing de software en la Universidad Cooperativa de Colombia Sede Popayán. In Memorias de Congresos UTP (pp. 149-154).

[48] Sbarski, P., & Kroonenburg, S. (2017). Serverless architectures on AWS: with examples using Aws Lambda. Simon and Schuster.

[49] J. Thönes, "Microservices," in IEEE Software, vol. 32, no. 1, pp. 116-116, Jan.-Feb. 2015, doi: 10.1109/MS.2015.11.

[50] Dragoni, N. et al. (2017). Microservices: Yesterday, Today, and Tomorrow. In: Mazzara, M., Meyer, B. (eds) Present and Ulterior Software Engineering. Springer, Cham. https://doi.org/10.1007/978-3-319-67425-4_12

[51] A. Saboor, A. K. Mahmood, M. F. Hassan, S. N. M. Shah, F. Hassan and M. A. Siddiqui, "Design Pattern Based Distribution of Microservices in Cloud Computing Environment," 2021 International Conference on Computer & Information Sciences (ICCOINS), Kuching, Malaysia, 2021, pp. 396-400, doi: 10.1109/ICCOINS49721.2021.9497188.

[52] X. J. Hong, H. Sik Yang and Y. H. Kim, "Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application," 2018 International Conference on Information and Communication Technology Convergence (ICTC), Jeju, Korea (South), 2018, pp. 257-259, doi: 10.1109/ICTC.2018.8539409.

[53] R. A. P. Rajan, "Serverless Architecture - A Revolution in Cloud Computing," 2018 Tenth International Conference on Advanced Computing (ICoAC), Chennai, India, 2018, pp. 88-93, doi: 10.1109/ICoAC44903.2018.8939081.

[54] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). Experimentation in software engineering. Springer Science & Business Media.