Cloud-Native Architecture for Distributed Systems that Facilitates Integration with AIOps Platforms

Juan Pablo Ospina Herrera(✉) and Diego Botia

Maestría en ingeniería, Universidad de Antioquia, Medellín, Colombia
{juan.ospina3,diego.botia}@udea.edu.co

Abstract. DevOps has significantly enhanced application operations through the utilization of containers and CI/CD. It still relies on human intervention in the event of failures in any system component. Many existing solutions are limited to specific issues, such as reacting to server outages and scaling them up. As the complexity of distributed systems continues to grow due to the simultaneous operation of numerous components, even minor unavailability can substantially impact application reliability and result in significant economic consequences for businesses. Therefore, it is imperative that the solutions being developed minimize risks and increasingly automate these operations. In light of these challenges, the emergence of AIOps offers a promising solution using artificial intelligence techniques, including machine learning and big data, to operate and maintain application infrastructures, reduce operational complexity, and automate IT operations processes. Implementing such solutions has been shown to improve system quality and significantly reduce the time it takes to detect errors and recover from them. These advancements mark significant progress in the realm of operations. However, despite these benefits, widespread adoption of AIOps solutions by most companies remains limited due to the challenges associated with implementing them in large projects and the lack of clear integration pathways for emerging solutions. In this paper, we propose a holistic architecture that facilitates the integration of cloud-native distributed systems with these new solutions.

Keywords: Software Architecture · Distributed Systems · AIOps · Cloud-Native

1 Introduction

In distributed environments, maintaining multiple services 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 (Average time to detect an error) can be reduced from 10 min to 1 min and the MTTR (Average time to repair an error) can be reduced from 60 min to 30 s [1] which shows the great power that we can achieve by implementing this type of technology.

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.

Currently, there are specialized solutions that target specific aspects of IT operations management by employing artificial intelligence techniques (particularly within AIOps subcategories). However, there is a lack of clear guidance on effectively harnessing these tools to establish an administration framework based on AIOps. This challenge is especially prominent in the context of distributed systems, where the simultaneous operation of multiple services and the failure of a single component can significantly impact the proper functionality of the system. Consequently, a comprehensive approach is needed to address these complexities and ensure the seamless integration of AIOps principles in managing distributed systems.

Several prior research works have primarily addressed individual tasks or specific subareas within the realm of AIOps [2–4]. The majority of these studies primarily focus on algorithms [1], such as anomaly detection [5–7], clustering analysis [8], failure prediction [9–11], and cost optimization [12]. This motivates the need for a study focused on investigating the main architectures used in case studies of systems managed by AIOps solutions. This research will serve as a foundation for systems to migrate to this type of self-managed administration or new systems that want to take advantage of these powerful solutions.

To build this architecture, 45 articles were analyzed to identify the key patterns, principles, and tools used for integration with AIOps platforms. Next, critical points were identified, and the main patterns for distributed systems were searched to build the proposed architecture, using the C4 model for its presentation. Finally, this new architecture was evaluated through one use case, measuring the MTTD and MTTR times to assess its successful integration with the AIOps platform, along with five specific metrics to evaluate the facility provided by this architecture.

2 Literature Review

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 science papers or books.

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 review.

Table 1. Patterns, principles, and tools used to integrate with AIOps platforms.

Article	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 [13]	Layered	Y	Monitoring, Virtualization	Docker, Kubernetes, OpenStack, AWS, MySQL
An Anomaly Detection Algorithm for Microservice Architecture Based on Robust Principal Component Analysis, 2019 [14]	Microservice, Reactive	Y	Synchronous communication, Middleware, Containerization, Logging, Monitoring	Docker, Oracle
AI-Governance and Levels of Automation for AIOps- supported System Administration, 2019 [15]	N/A	Y	Virtualization, Load balancing, CI/CD, Self-healing, Self-stabilizing	Kubernetes
Ananke: A framework for Cloud-Native Applications smart orchestration, 2020 [16]	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. Chap. 3: Application Optimisation: Workload Prediction and Autonomous Autoscaling of Distributed Cloud Applications, 2020 [17]	Client-server, Cloudlet, Service-oriented, Microservices	Y	API gateway, REST, Load balancing, Autoscaling, Remediation, Virtualization, Monitoring	RECAP

Table 1 presents a comparison of the information found in the selected articles summarizing the patterns, principles, and tools used in each one. 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. There are three critical sections that are commonly found in all the papers:

- 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.

After collecting and analyzing the list of patterns for each of these three critical sections, we can detect the similarity that exists among many of them. 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. 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 (Table 2).

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.

For the middleware patterns, we can select only one and it will be enough. The best pattern may vary, but the API gateway is easy to implement than the service mesh pattern. The asynchronous messaging pattern is only applicable

Table 2. Recommended patterns.

Section	Recommended	Discarded
Monitoring	Health check API, log aggregation, distributed tracing, exception tracking, application metrics, microservices chassis, sidecar	
Middleware	API Gateway, Service Mesh, Asynchronous messaging	Replicated load-balanced services, Ownership election, Adapters
Virtualization	Kubernetes	Virtual machines, Containers

for specific use cases. Replicated load-balanced services, ownership election, and adapters are patterns already implemented in tools like Kubernetes.

The big winner in the virtualization section is Kubernetes. 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.

3 Proposed Architecture

Based on the previous analysis, a general architecture is proposed that is easy to adapt and implement by developers working with distributed systems. 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 focused on a certain perspective of the system [18]. It is based on a hierarchy of four levels: system context, container, component, and code. Each level provides a different level of detail and abstraction.

3.1 System Context Level

At the system context level, there is not 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 is 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 is crucial to understand the business requirements and the stakeholders of the system to determine the appropriate boundaries (Fig. 1).

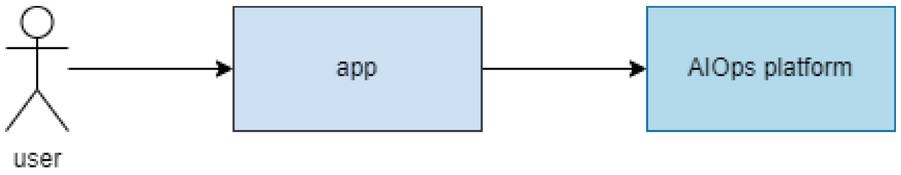

Fig. 1. System context view.

3.2 Container Level

At this level, we applied some patterns such as sidecar and asynchronous messaging. We also have a container dedicated to middleware and represent the monitoring systems.

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.

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. The sidecar pattern can be customized depending on the use case, but Kubernetes provides Envoy Proxy to implement it easily and natively within Kubernetes.

In Fig. 2, 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.

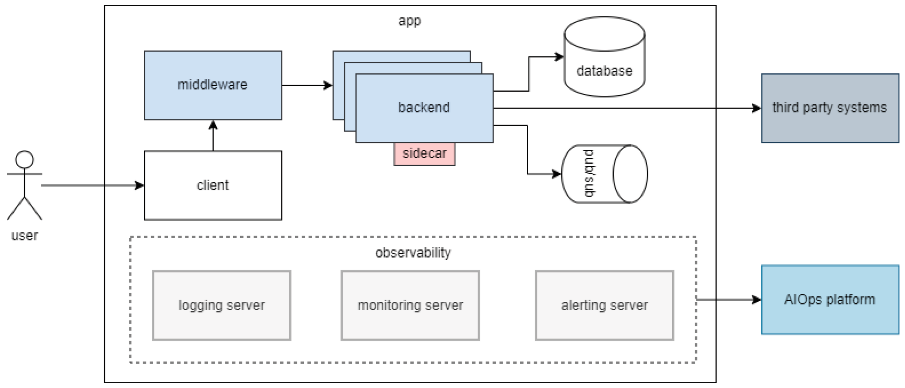

Fig. 2. Container view.

3.3 Component 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 (Fig. 3).

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.

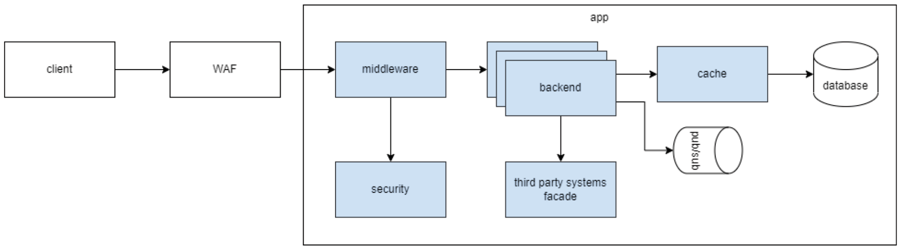

Fig. 3. Component view.

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 [19].

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.

3.4 Code Level

We cannot 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.

4 Validation and Results

As AIOps Platform we will use DevOps Guru which is an AWS service that helps developers and IT teams improve application reliability and performance [20]. 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. In that case we only configure it to take the logs generated on CloudWatch and generate an alert to create new instances before the error occurs.

We tested the architecture with one use case and 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 ability of the user to order food for delivery.

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 they do not change with respect to the one shown in the proposed architecture (Fig. 4).

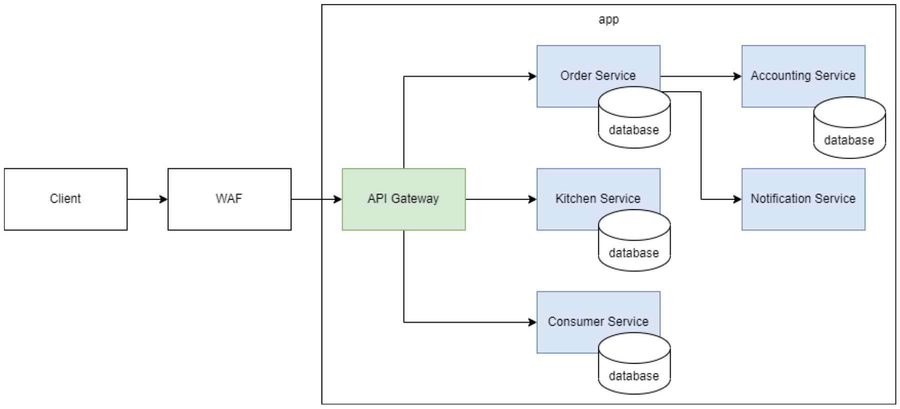

Fig. 4. Component view for the order food delivery application.

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. All microservices have a similar architecture at the code level, so we are going to diagram the Order service which is the main one for this business model.

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, we shared the same base code for them. It also includes the configuration to enable a health check endpoint (Fig. 5).

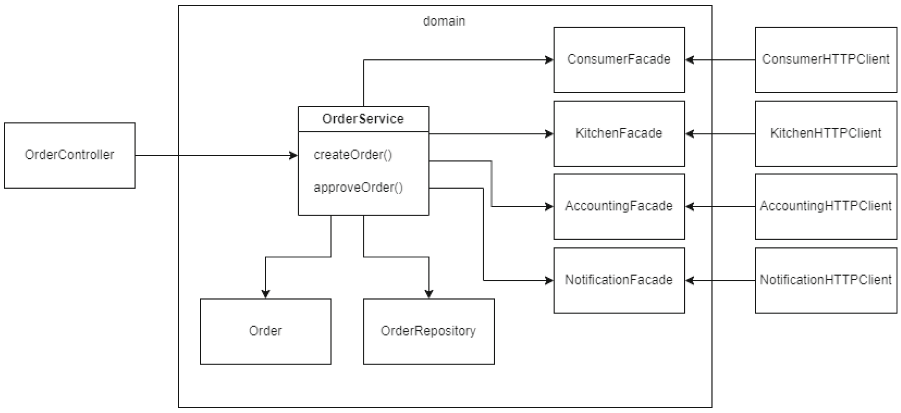

Fig. 5. Code view for the Order service.

This 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 using JMeter 5.5 were carried out on the order microservice, simulating 100 concurrent users trying to create new orders for 22 min. For this test, errors were also injected into all microservices every 5 min, 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.

The percentage of failed requests was 0.492% of 1159040 requests executed, 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 (Table 3).

Table 3. 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.

$$Average\ downtime = \frac{\sum_{0}^{num\ of\ restarts}(first\ successful\ request - first\ failed\ request)}{number\ of\ restarts} \quad (1)$$

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.

$$MTTD = \frac{\sum_{0}^{num\ of\ restarts}(time\ insight\ created\ -\ time\ first\ failed\ request)}{number\ of\ restarts} \quad (2)$$

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.

$$MTTR = \frac{\sum_{0}^{num\ of\ restarts}(time\ first\ successful\ request-\ time\ insight\ created)}{number\ of\ restarts} \quad (3)$$

Considering the benchmark values obtained in the state of the art for MTTD times of less than 1 min and MTTR of less than 30 s, 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.

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 [21]. 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:

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. **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 s, demonstrating its robustness. **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. See Fig. 6.

For interoperability, the following measures were considered: **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. **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. See Fig. 6.

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 [22] 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 times. 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 to measure the flexibility.

The average integration time spent was 44.8 min. 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. The average time spent on flexibility was 24.7 min. 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. Both results show that in a few minutes the architecture can be integrated and made changes (flexible).

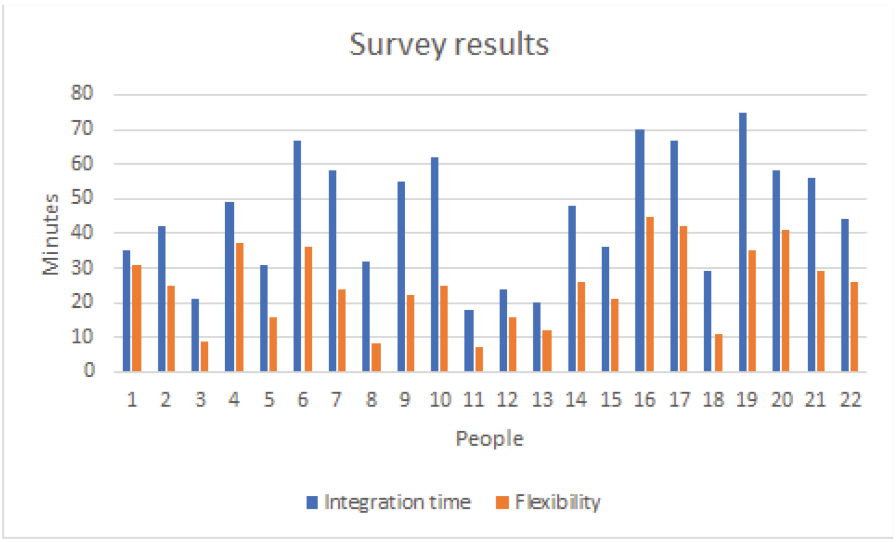

Fig. 6. Survey results for integration time and flexibility.

5 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, in which the use of monitoring patterns, middleware, and virtualization. Although the validation with the use case required extensive analysis, the benefits provided by the proposed architecture were successfully evaluated. A low error rate of less than 1% was obtained, and error detection and correction times were below the standards (MTTD of 1 min and MTTR of 30 s). Based on the results of the 5 metrics obtained for the use case, it can be inferred that the architecture implemented in this use case is adaptable and interoperable. As a result, we can conclude the proposed architecture facilitates integration with AIOps platforms.

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 middleware, 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.

References

1. Shen, S., Zhang, J., Huang, D., Xiao, J.: Evolving from traditional systems to AIOps: design, implementation and measurements (2020)
2. Notaro, P., Cardoso, J., Gerndt, M.: A systematic mapping study in AIOps. In: Hacid, H., et al. (eds.) ICSOC 2020. LNCS, vol. 12632, pp. 110–123. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76352-7_15
3. Kobbacy, K.A., Vadera, S., Rasmy, M.H.: Ai and or in management of operations: history and trends. J. Oper. Res. Soc. **58** (2007)
4. Mukwevho, M.A., Celik, T.: Toward a smart cloud: a review of fault-tolerance methods in cloud systems. IEEE Trans. Serv. Comput. (2018)
5. Li, L., Hansman, R.J., Palacios, R., Welsch, R.: Anomaly detection via a gaussian mixture model for flight operation and safety monitoring (2016)
6. Farshchi, M., Schneider, J.G., Weber, I., Grundy, J.: Metric selection and anomaly detection for cloud operations using log and metric correlation analysis (2017)
7. Nedelkoski, S., Cardoso, J., Kao, O.: Anomaly detection from system tracing data using multimodal deep learning (2019)
8. Du, M., Versteeg, S., Schneider, J.G., Han, J., Grundy, J.: Interaction traces mining for efficient system responses generation (2015)
9. Salfner, F., Lenk, M., Malek, M.: A survey of online failure prediction methods (2010)
10. Wang, Z., et al.: Failure prediction using machine learning and time series in optical network (2017)
11. Wang, H., Zhang, H.: AIOps prediction for hard drive failures based on stacking ensemble model (2020)
12. Chen, Q., Zheng, Z., Hu, C., Wang, D., Liu, F.: Data-driven task allocation for multi-task transfer learning on the edge (2019)
13. Mormul, M., Stach, C.: A context model for holistic monitoring and management of complex it environments (2020)
14. Levin, A.: An anomaly detection algorithm for microservice architecture based on robust principal component analysis (2019)
15. Gulenko, A., Acker, A., Kao, O., Liu, F.: AI-governance and levels of automation for AIOps-supported system administration (2019)
16. Di Stefano, A., Di Stefano, A., Morana, G.: Ananke: a framework for cloud-native applications smart orchestration (2020)
17. Cardoso, J., Kao, O.: Application optimisation: workload prediction and autonomous autoscaling of distributed cloud applications (2020)
18. Vázquez-Ingelmo, A., García-Holgado, A., García-Peñalvo, F.J.: C4 model in a software engineering subject to ease the comprehension of UML and the software (2020)
19. Noah, P., Seman, S.: Distributed multi-level query cache: the impact on data warehousing. Issues Inf. Syst. (2012)
20. Sawant, N., Sengamedu, S.H.: Learning-based identification of coding best practices from software documentation (2022)
21. Paz, J.A.M., Gomez, M.Y.M., Rosas, S.C.: 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 (2017)
22. Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A.: Experimentation in Software Engineering. Springer Science & Business Media, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29044-2