

Edge Anomaly Detection Framework for AIOps in Cloud and IoT

Pieter Moens¹^a, Bavo Andriessen¹, Merlijn Sebrechts¹^b, Bruno Volckaert¹^c
and Sofie Van Hoecke²^d

¹*Internet and Data Science Lab (IDLab), Ghent University-imec, Ghent, Belgium*
{pieter.moens, bavo.andriessen, merlijn.sebrechts, bruno.volckaert, sofie.vanhoecke}@ugent.be

Keywords: AIOps, Cloud Computing, Internet of Things, Microservices, Anomaly Detection, Monitoring

Abstract: Artificial Intelligence for IT Operations (AIOps) addresses the rising complexity of cloud computing and Internet of Things by assisting DevOps engineers to monitor and maintain applications. Machine Learning is an essential part of AIOps, enabling it to perform Anomaly Detection and Root Cause Analysis. These techniques are often executed in centralized components, however, which requires transferring vast amounts of data to a central location. This increase in network traffic causes strain on the network and results in higher latency. This paper leverages edge computing to address this issue by deploying ML models closer to the monitored services, reducing the network overhead. This paper investigates two architectural approaches: a sidecar architecture and a federated architecture, and highlights their advantages and shortcomings in different scenarios. Taking this into account, it proposes a framework that orchestrates the deployment and management of distributed edge ML models. Additionally, the paper introduces a Python library to assist data scientists during the development of AIOps techniques and concludes with a thorough evaluation of the resulting framework towards resource consumption and scalability. The results indicate up to 98.3% reduction in network usage depending on the configuration used while maintaining a minimal increase in resource usage at the edge.

1 INTRODUCTION


The rise in cloud computing and the (Industrial) Internet of Things (IoT) introduced a shift in modern application architectures. Container-based architectures, e.g., the microservice architecture, tackle the shortcomings of the traditional monolithic architecture such as scalability (Dragoni et al., 2017). The distributed nature of the microservice architecture increases the complexity of the system, which in turn increases the importance of monitoring and observability. Artificial Intelligence for IT Operations (AIOps) aims to support operations engineers to maintain the applications and avoid system downtime by leveraging Machine Learning (ML) for e.g. Anomaly Detection (AD) and Root Cause Analysis (RCA) (Dang et al., 2019).


Monitoring and anomaly detection are commonly performed in a centralized manner, i.e., one service is responsible for collecting and processing the monitoring data from all services in the distributed sys-


tems. While this approach is easily manageable, it has a major drawback: all monitoring data must traverse the network to reach the central monitoring service, therefore introducing additional network traffic and increasing network latency. Reducing this network traffic can reduce costs and improve the performance of the network in general, especially in public cloud environments (e.g. Amazon Web Services, Google Cloud Platform, Microsoft Azure) and IoT use-cases.


To address this increase in network usage, edge ML has been introduced. It brings intelligence close to the services they are monitoring, which are located at the edge of the infrastructure layer. In order to achieve this, several challenges should be addressed. Firstly, to deploy ML models on edge devices, they must be optimized towards resource usage (e.g. CPU and memory consumption). Secondly, moving from a centralized monitoring and AIOps solution towards distributed learning at the edge introduces an increase in complexity with regards to orchestration of the distributed ML models and services (Goethals et al., 2021).

This paper presents Deucalion, an edge anomaly detection framework for cloud computing and IoT applications. According to Greek mythology, Deu-

^a <https://orcid.org/0000-0003-2035-8766>

^b <https://orcid.org/0000-0002-4093-7338>

^c <https://orcid.org/0000-0003-0575-5894>

^d <https://orcid.org/0000-0002-7865-6793>

calion, son of Prometheus, is able to survive the flood by building an ark. Analogous, the Deucalion framework enables DevOps engineers to survive the modern flood of monitoring data. It leverages edge computing and distributed learning to provide support for the automated deployment of ML models on the edge, real-time data collection from the monitored resources. In addition to the presented framework to orchestrate the deployment and management of the distributed edge ML models, a supporting Python library has been implemented to ease the development process of the models and to bridge the gap between data scientists and DevOps engineers.

The remainder of this article is structured as follows: Firstly, Section 2 discusses the state-of-the-art in edge computing frameworks, Section 3 provides background information about the service mesh pattern for microservices and highlights its advantages, Section 4 describes the methodology and architectures used in our proposed solution, Section 5 discusses design choices during implementation of the framework and resulting Python library, Section 6 evaluates the presented framework with regard to overhead for the monitored system and finally, Section 7 concludes this paper and outlines a vision for future work.

2 RELATED WORK

Edge computing is an active research area, particularly in the ML department. Thorough research is done on how models can be reduced and optimized to be deployed at the edge. The focus of this paper is therefore not the optimization of the machine learning or anomaly detection models themselves, but an optimization of the way these models can be deployed in a decentralized manner, such that the network usage caused by monitoring is reduced.

(Calo et al., 2017) identifies three generic architectures for applying AI to IoT (AIoT) in the edge. First, a centralized approach where the AI is applied in a centralized cloud environment. All data are sent to and processed by the cloud. Secondly, an edge approach where the data is processed in the edge nodes. The third approach is an enhancement of the edge approach with a tree-structured architecture: the data is produced in the leaf nodes but processed in the internal nodes of the tree structure. The data is further aggregated as it goes up the tree.

(Debauche et al., 2020) describe a hybrid architecture for edge AIoT by making a distinction between three types of nodes in the infrastructure layer: (i) remote nodes in the cloud, (ii) nodes in the edge and (iii)

IoT nodes, or devices. The proposed architecture consists of a centralized cluster of nodes in the cloud and one or more (micro) clusters at the edge. The data are collected on IoT devices or nodes and pushed to the edge clusters. These edge clusters host numerous services including machine learning models, databases and a streaming platform. The processed data from the different edge clusters is then collected and aggregated in the cloud. All nodes in the system are managed by Kubernetes container orchestration and the ML models are deployed on the cloud and edge nodes as containerized applications. The proposed solution yields promising results with regard to latency and network usage reduction, but the clusters at the edge introduce a significant overhead in terms of resource usage.

(Becker et al., 2020) discusses preliminary ideas towards AIOps for edge computing including the automated deployment of AD models on edge devices. They introduce a framework, ZerOps4E, which leverages the Bitflow stream processing framework for both the collection of data streams and the deployment of the models. They specifically target lightweight edge devices capable of running Docker containers (e.g. Raspberry Pi). The authors underline the need for an edge anomaly detection framework, but they do not provide enough details and insights in the architecture and implementation of the framework to enable reproduction of or extensions to their proposed solution, making it difficult to evaluate or build upon the presented work.

(Raj et al., 2021) present an automation framework for edge MLOps. They describe a continuous integration and development pipeline for ML models at the edge and focus particularly on challenges such as online learning, analytics and versioning. Similar to previously mentioned work, they leverage Docker containerization for deployment of the ML models on different types of devices. Their framework is based on the Azure cloud environment, using Azure ML Services, Azure DevOps, Azure IoT central and Azure blob storage, which imposes vendor lock-in.

In summary, recent research has shed light on the benefits of, and need for, AIOps in cloud computing and IoT. Although various architectures are proposed to optimize resource consumption, little attention is given to the orchestration and deployment of the distributed models as well as the extendibility and flexibility of the solution to different scenarios. This paper tackles these shortcomings by presenting a single framework for AIOps that focuses on automated deployment and data collection under various configurations, without altering the underlying application.

3 BACKGROUND: SERVICE MESH

Cloud native applications can be designed as distributed (collections of) microservices. Each microservice or collection thereof is tasked with a discrete business function. Interaction by the users with the application often propagates throughout the system as these microservices communicate with one another. As the number of microservices in the application grows larger, monitoring the network traffic becomes complex. A service mesh is a dedicated infrastructure layer that can be added to the application to enable features such as observability, traffic management, and security (Li et al., 2019). It divides the application in two layers. Firstly, the data plane which consists of the application specific microservices and a number of network proxies. These proxies guide and manage all service-to-service communication. Secondly, the control plane which consists of services dedicated to performing the service mesh specific tasks such as managing and configuring the network proxies.

A number of service mesh technologies such as Amazon Web Services (AWS) App Mesh¹, Linkerd², AirBnB Synapse³ and Istio (Calcote and Butcher, 2019) have been developed over the last years. All of these technologies focus on network traffic management (e.g. load balancing) and monitoring (e.g. network tracing). Istio is an open-source service mesh technology based on Kubernetes container orchestration. Its control plane consists of three services: (i) the pilot, which communicates with the proxies to handle traffic management and routing, (ii) the citadel, which provides secure communication between services and (iii) the galley, which is responsible for configuration management, distribution and processing. Istio deploys the network proxies as sidecar containers (Sheikh et al., 2018). By doing this, the service mesh can be seamlessly applied to any cloud native application without altering the underlying microservices. A sidecar proxy is a containerized application that runs on the same device as the service it is monitoring. This means that communication between the proxy and the service is local and does not influence the network.

¹<https://aws.amazon.com/app-mesh/>

²<https://linkerd.io/>

³<https://airbnb.io/projects/synapse/>

4 ARCHITECTURE

In this paper, we discuss and showcase how the concept of a service mesh can be expanded, beyond the use case of network traffic management and monitoring, towards AIOps and edge anomaly detection. The concept of sidecar containers is not restricted to network proxies. They can be any type of logical component that should be deployed closely to the application service, such as an anomaly detection model. Similarly to the previously mentioned service mesh technologies, a control plane can be defined that consists of framework specific services. The Deucalion control plane consists of:

- **Sidecar Injector:** A component that handles injection of the anomaly detection sidecar in all microservices of the application.
- **Alert Manager:** A centralized service that collects the output of the deployed models, such as detected anomalies.

As these control plane services are located in the cloud, all communication between the deployed sidecar containers and the control plane components is regarded as overhead in network traffic introduced by the monitoring solution. We present and discuss two approaches to optimize the reduction in network traffic and latency by moving the ML models and monitoring solutions to the edge while minimizing the overhead introduced by the framework, such as resource usage at the intelligent edge.

4.1 Sidecar Architecture

The sidecar architecture minimizes the network traffic by placing the ML models as close to the monitored services as possible. Each deployed Kubernetes pod containing an application service will be injected with an additional sidecar container. This sidecar container pulls the monitoring data from the pod in which it is deployed. As a result, all monitoring data is contained within the pod itself and only detected events or anomalies are pushed towards the centralized *Alert Manager*. This architecture maximizes the reduction in the network traffic and can optimally be used in a scenario where the edge devices have sufficient resources and network communication can become costly (i.e. 5G networks or public cloud environments). A schematic overview of the proposed sidecar architecture is shown in Figure 1.

4.2 Federated Architecture

Running a sidecar container alongside each application service introduces an increase in resource usage,

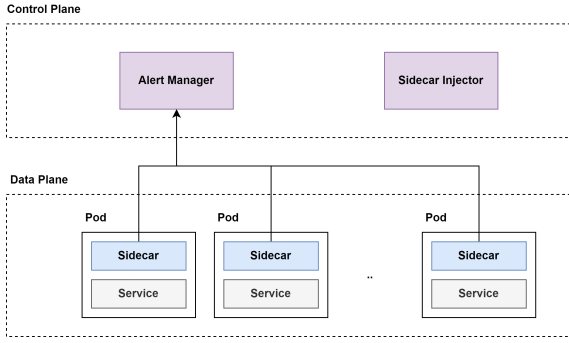


Figure 1: Sidecar architecture where a sidecar container is injected into every pod in the data plane. The control plane consists of an Alert Manager, which receives the output of the anomaly detection models, and a Sidecar Injector.

i.e., memory and CPU consumption. In some use-cases, it might make sense to group application services in federations. For example, given a scenario where a cloud application is deployed across multiple clusters or availability zones, services within the same availability zone could be considered to be in the same federation. Similarly, in an IoT application, services within the same physical boundary or local network, e.g. the same building, could represent a single federation. A visual representation of the proposed federated approach is shown in Figure 2.

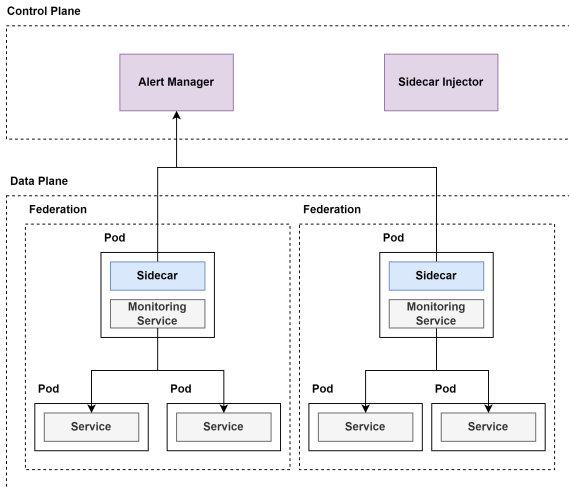


Figure 2: Architecture of the federated approach where the data plane is divided into federations of data plane services. Only one monitoring service and anomaly detection model is deployed per federation.

In this approach, hierarchical (sub)layers are created. A monitoring solution (e.g. Prometheus) is deployed within each federation. This solution functions as a centralized monitoring service within the federation and collects all monitoring data from the application services, or targets, in its federation. An

ML service is then deployed as a sidecar next to the monitoring instances which it queries to receive all collected data. This way, the overhead on the system with regard to resource usage can be heavily reduced in exchange for a lesser reduction in network usage as the monitoring service collects monitoring data from all monitored services or pods. Depending on the use-case, the size of each federation can be determined in order to optimize the overall cost of the application.

5 IMPLEMENTATION

To implement the methodology discussed in the previous section, a number of technology choices can be made. The most common monitoring solution for cloud applications today is Prometheus. Prometheus is an open-source time-series database that collects data from its targets using a pull policy. It relies on all services in the application to expose data in the form of metrics on a configured HTTP(S) endpoint. Prometheus uses a standardized data format, OpenMetrics, that is well documented and widely accepted within the community (Di Stefano et al., 2021).

The proposed solution is twofold. Firstly, a Python framework has been designed to support data scientists in developing real-time AD services without requiring expertise in cloud native applications or different infrastructures. Secondly, the control plane components consisting of two components: the sidecar injection, which handles orchestration of the AD services and ensures automation and ease of deployment for DevOps engineers, and the central component to dynamically collect the output of the deployed ML models. For this central component, the choice for Prometheus Alert Manager has been made. The Alert Manager is open-source and scalable by design. It is a mature component and supports out-of-the-box integration with various consumers. As the Prometheus Alert Manager is well-documented, in-depth functionality will not be elaborated on in this paper. We refer the reader to (Sabharwal and Pandey, 2020) for all documentation about this component.

5.1 Python Framework

To establish real-time AD, the monitoring data should be continuously collected or received from the chosen monitoring solution and exposed or pushed to the central alert manager. In order to relieve the data scientist of this implementation effort, a framework is created that ensures integration with the applications services and the control plane components.

As mentioned in Section 4, the strategy to col-

lect data differs based on the deployed architecture. When using the sidecar architecture, the data is directly collected from the application services. Similarly to a centralized Prometheus solution, all application services expose their monitoring data or metrics on a dedicated endpoint. The AD service then periodically scrapes the configured endpoint of the service it is monitoring. For the federated architecture, however, a Prometheus instance is deployed for each federation. Prometheus is designed to handle data collection from multiple targets. The AD service then only needs to query the collected data from the Prometheus instance.

The core design of the framework is based on the observer pattern. The developed model can be registered as an observer. By doing this, the model is notified whenever a new (batch of) data point(s) is available and the prediction pipeline can be executed. When an anomaly is detected, the observer creates an event which is pushed to the central Alert Manager. A visualization of the different modules is shown in Figure 3.

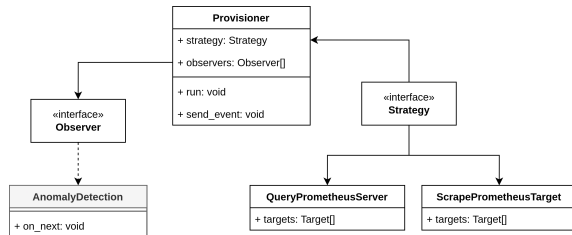


Figure 3: Architecture of the Deucalion framework. The *AnomalyDetection* class is an implementation of the *Observer* interface. This component is created by the user of the framework and is use-case specific.

To prevent vendor lock-in, the framework is designed with modularity in mind and can be easily extended for different technology design choices, allowing components such as Prometheus and the Alert Manager to be exchanged for alternative solutions.

5.2 Sidecar Injection

The *Sidecar Injector* is implemented as a Kubernetes *Mutating Admission Webhook* (Kubernetes, 2022). An Admission Webhook in Kubernetes implements an HTTPS callback which is notified whenever a resource is created, updated or deleted. As the name suggests, the webhook is able to mutate the newly created or updated resource before it is deployed. During this stage, the sidecar container is dynamically injected.

To simplify the configuration and deployment of the framework, Helm charts have been created

(Howard, 2022). With a limited set of configuration values, the framework can be installed on any Kubernetes cluster (*version* \geq v1.16) with the desired sidecar or federated architecture. With a single command, the framework takes care of the deployment of all control plane services as well as additional resources required for a Kubernetes Admission Webhook, such as SSL/TLS certificates and Role-based Access Control (RBAC).

For the sidecar injection, two arguments are required: (i) the image of the sidecar container and (ii) a ConfigMap containing sidecar specific configuration. These arguments can be configured for each deployment in the application.

```

kind: Deployment
metadata:
  annotations:
    deucalion-config-map:
      deucalion-sidecar-config-map
    deucalion-sidecar-image:
      "pimoens/deucalion-app:latest"
  
```

Flexibility towards the various strategies that can be adopted is assured through different configuration options for various resources in the Kubernetes manifests. The easiest method is to enable sidecar injection at the namespace level. A *namespace* is an abstract hierarchical layer within a Kubernetes cluster. It consists of a group of resources (e.g. pods, services, ingress). Using this method, all pods within the namespace are injected with a sidecar container. The configuration and sidecar image are specified at the namespace level, but can be overwritten for different deployments in the system.

```

kind: Namespace
metadata:
  labels:
    deucalion-injection: 'enabled'
  
```

To achieve the federated architecture, a *FederationID* is specified. Analogous to the sidecar architecture, this can be achieved at deployment or namespace level. In addition to sidecar injection, the framework will deploy and automatically configure a Prometheus instance per federation. The Prometheus instance monitors all containers in the namespace that have the *prometheus.io/scrape* annotation.

```

kind: Deployment
metadata:
  annotations:
    deucalion-federation: federation1
  
```

6 EVALUATION

The presented framework is evaluated using a benchmark microservice application: the Bookinfo applica-

tion developed by Istio⁴. The evaluation data is collected by deploying the demo on the Fed4Fire testbed (Demeester et al., 2016). A Kubernetes cluster consisting of four nodes is used, each node has a 2 Intel Xeon E5620 quad CPUs and 12GB RAM. With Intel Hyper-Threading enabled, this results in 16 logical CPUs per node.

To represent the federated architecture, the canary deployment with different versions of review service is considered as a single federation. The other services belong to a second federation. This approach represents a federated architecture where federations can be dynamically established when services are horizontally scaled. The manifests for both the sidecar architecture and federated architecture, as well as the captured evaluation data, are made available on GitHub⁵.

6.1 Network usage

To evaluate the benefits of using the presented framework, the reduction in network traffic is captured. Figure 4 shows the network usage (*mean transmitted bytes*) for the Bookinfo application when deployed using a centralized monitoring solution (i.e. Prometheus), the sidecar architecture and the federated architecture respectively. To showcase the baseline, the microservice application without monitoring is considered.

As expected the network traffic is heavily reduced when comparing the sidecar architecture with the classic centralized monitoring solution. The number of transmitted bytes per second for the centralized solution is around 56 KB/s, which results in a 55 KB/s, or 98.3%, reduction in network overhead, caused by a centralized monitoring solution such as Prometheus, when using the sidecar architecture. A reduction of 27.5% can also be noticed when comparing the federated architecture to the centralized solution. This reduction of network traffic is influenced by the number of federations used and their size. The optimal configuration is use-case specific and can be easily modified with the framework as discussed in Section 5. When comparing to the deployment without monitoring, the sidecar architecture deployment increases the network usage by 10.7%. This increase is introduced by the communication between the control plane components and the sidecars, i.e. one message per detected anomaly. For the deployed application during evaluation, this results in one message per 5 seconds.

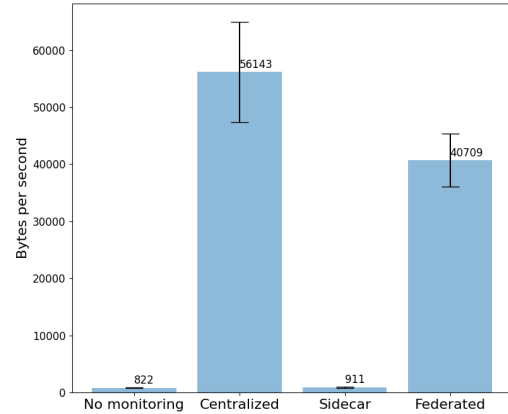


Figure 4: Mean transmitted bytes (Bps) captured for a deployment without monitoring, with centralized monitoring, using the sidecar architecture and using the federated architecture.

Quality of Service (QoS) and low latency are a priority in modern cloud native applications. Therefore, the impact of the framework on scalability is measured in addition to network usage. A load test is conducted on the different deployments using Locust (Pradeep and Sharma, 2019). The network performance (*total requests measured by Istio*) is shown in Figure 5. The results show that the maximum number of requests handled by the baseline without monitoring is higher than all three other approaches. A reduction of 4.8% can be measured in terms of maximum requests per second and a reduction of 1.2% can be measured in the average requests per second over the duration of the evaluation. Using a monitoring solution slightly impacts the scalability of the application. When comparing the three deployments with monitoring however, the measurements do not differ significantly.

6.2 Resource usage

As discussed in Section 4, the choice of architecture impacts the resource usage of the application. To evaluate the overhead introduced by the framework, cAdvisor (Casalicchio and Perciballi, 2017) is used to collect the CPU and memory usage in different deployments. The results shown in Figure 6 indicate that the sidecar architecture introduces no significant CPU overhead. When using centralized Prometheus monitoring, an increase of 10% can be noticed. This increase is comparable to the 8% increase when using the federated architecture. Both the centralized approach and the federated architecture are deployments

⁴<https://istio.io/latest/docs/examples/bookinfo/>

⁵<https://github.com/predict-idlab/deucalion>

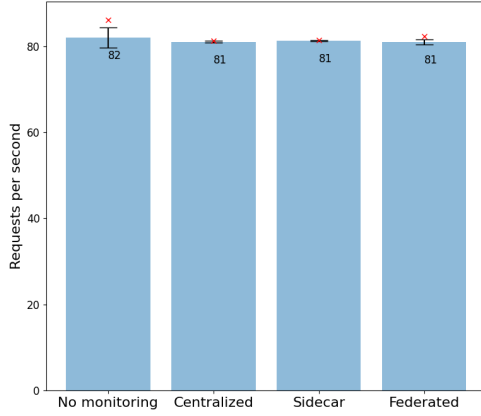


Figure 5: Scalability evaluation capturing the latency of the requests during a load test.

where targets are scraped by a Prometheus server instance, whereas in the sidecar architecture, targets are scraped by the Deucalion sidecar. This increase could therefore be assigned to the Prometheus server which implements extra functionality, e.g., health checks.

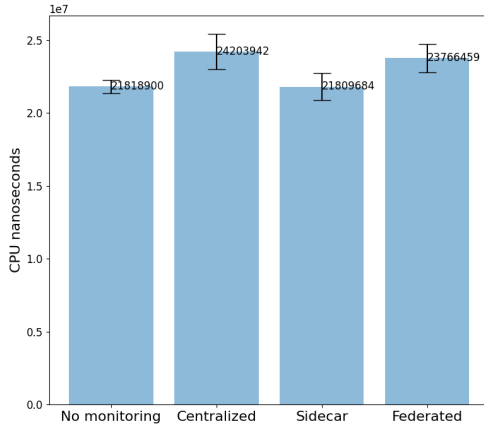


Figure 6: Comparison in total CPU usage of the services between a deployment without monitoring, with centralized monitoring, using the sidecar architecture and using the federated architecture.

To provide an indication of the Deucalion sidecar container, a comparison is made with the Envoy proxy sidecar used by Istio. The results are shown in Figure 7. The average CPU time used by Deucalion sidecars is 3.03 (± 0.572) milliseconds per second while the Envoy proxies consume 21.8 (± 0.467) milliseconds per second. The applications at rest consumed 16.5 (± 5.16) milliseconds per second on aver-

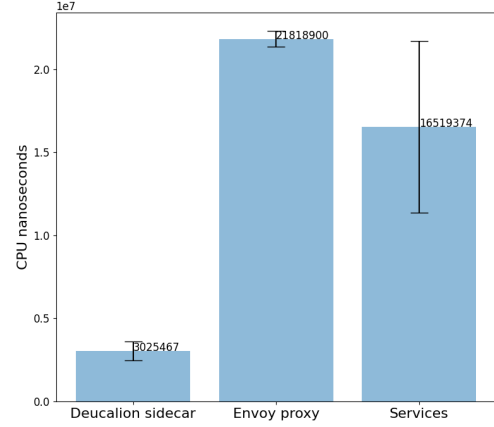


Figure 7: Comparison in CPU usage between the Deucalion sidecar container and the Envoy proxy used by Istio. To give an indication of the overhead on the system, the CPU usage for the application services is shown.

age, which is 5 times as much as the Deucalion sidecars. This result shows that the impact of the framework itself is considerably low when compared to the CPU time consumed by the Envoy proxy and the application. It should be noted that the deployed AD service for the demo is a minimal application. Resource usage will increase as the developed ML models increase in complexity.

7 CONCLUSIONS AND FUTURE WORK

This paper proposes two distinct architectures for edge anomaly detection and distributed learning to reduce the overhead in network usage and latency introduced by AIOps in cloud computing and IoT applications: (i) a sidecar architecture and (ii) a hierarchical, federated architecture. It evaluates both approaches towards resource consumption and scalability. The sidecar architecture yields the highest reduction of network traffic compared to state-of-the-art centralized monitoring solutions, up to 98.3%, as it deploys ML models as close as possible to the monitored services and therefore minimizes the network usage. Due to the ML models being deployed on the edge devices, however, it introduces an increase in resource usage. The sidecar architecture can therefore be optimally applied to a scenario where the edge devices have sufficient resources and communication over the network is costly. In case the sidecar architecture is not feasible, the federated architecture can be adopted. This architecture hierarchically groups

the edge devices in federations. For each federation, a monitoring service is deployed. The federated architecture can be configured with regard to the number of federations and the size of each federation, and therefore can be used to reduce the resource usage while still maximizing the reduction in network usage.

Based on the two architectures, this paper introduces the Deucalion framework to support data scientists and DevOps engineers during both the development and the deployment stages of the AIops models. This framework enables automated deployment and orchestration of ML models at the edge with minimal configuration overhead. Furthermore, a Python library is implemented to relieve data scientists of any integration effort with monitoring solutions during the development of real-time AD services.

The framework is based on Kubernetes container orchestration, which is commonly supported by many cloud providers, preventing a vendor lock-in. Due to the modular design of the framework, future work includes extensions for different technology design choices (e.g. Prometheus alternatives). Currently, the framework focuses specifically on development and deployment of the AD services. In the future, support for advanced MLOps can be integrated, enabling maintenance and versioning of the different deployed models. Collaboration between decentralized models, using federated learning, is a heavily researched field. The control plane of the presented framework can therefore be extended to support further orchestration between models and enable federated learning while maintaining the automated deployment at the edge.

REFERENCES

- Becker, S., Schmidt, F., Gulenko, A., Acker, A., and Kao, O. (2020). Towards aiops in edge computing environments. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 3470–3475. IEEE.
- Calcote, L. and Butcher, Z. (2019). *Istio: Up and running: Using a service mesh to connect, secure, control, and observe*. O’Reilly Media.
- Calo, S. B., Touna, M., Verma, D. C., and Cullen, A. (2017). Edge computing architecture for applying ai to iot. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 3012–3016. IEEE.
- Casalicchio, E. and Perciballi, V. (2017). Measuring docker performance: What a mess!!! In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 11–16.
- Dang, Y., Lin, Q., and Huang, P. (2019). Aiops: real-world challenges and research innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 4–5. IEEE.
- Debauche, O., Mahmoudi, S., Mahmoudi, S. A., Manneback, P., and Lebeau, F. (2020). A new edge architecture for ai-iot services deployment. *Procedia Computer Science*, 175:10–19.
- Demeester, P., Van Daele, P., Wauters, T., and Hrasnica, H. (2016). Fed4fire: the largest federation of testbeds in europe. In *Building the future internet through FIRE*, pages 87–109.
- Di Stefano, A., Di Stefano, A., Morana, G., and Zito, D. (2021). Prometheus and aiops for the orchestration of cloud-native applications in ananke. In *2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 27–32. IEEE.
- Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer.
- Goethals, T., Volckaert, B., and De Turck, F. (2021). Enabling and leveraging ai in the intelligent edge: A review of current trends and future directions. *IEEE Open Journal of the Communications Society*.
- Howard, M. (2022). Helm—what it can do and where is it going? *arXiv preprint arXiv:2206.07093*.
- Kubernetes (2022). Kubernetes documentation. <https://kubernetes.io/docs/home/>. Accessed: 2022-11-09.
- Li, W., Lemieux, Y., Gao, J., Zhao, Z., and Han, Y. (2019). Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 122–1225. IEEE.
- Pradeep, S. and Sharma, Y. K. (2019). A pragmatic evaluation of stress and performance testing technologies for web based applications. In *2019 Amity International Conference on Artificial Intelligence (AICAI)*, pages 399–403. IEEE.
- Raj, E., Buffoni, D., Westerlund, M., and Ahola, K. (2021). Edge mlops: An automation framework for aiots applications. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 191–200. IEEE.
- Sabharwal, N. and Pandey, P. (2020). Getting started with prometheus and alert manager. In *Monitoring Microservices and Containerized Applications*, pages 43–83. Springer.
- Sheikh, O., Dikaleh, S., Mistry, D., Pape, D., and Felix, C. (2018). Modernize digital applications with microservices management using the istio service mesh. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, pages 359–360.