SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Telemetry Driven Network Optimization for Edge-Cloud Orchestration Frameworks

Simon Zelenski

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Telemetry Driven Network Optimization for Edge-Cloud Orchestration Frameworks

Telemetriegesteuerte Netzwerkoptimierung für Edge-Cloud Orchestrierungsframeworks

Author:	Simon Zelenski
Examiner:	Prof. Dr.-Ing. Jörg Ott
Supervisors:	M.Sc. Giovanni Bartolomeo
	Dr. Nitinder Mohan
Submission Date:	15.07.2025

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.07.2025 Simon Zelenski

Acknowledgments

I owe my deepest gratitude to my family, whose unwavering support and encouragement have been the foundation of this academic journey. Their love and belief in me have carried me through every challenge.

To my parents, who have shaped me into the person I am today through their wisdom, sacrifices, and endless care.

To my girlfriend, Julie, for her constant encouragement and belief in me. Your support has meant more than words can express.

A heartfelt thank you to my friends, who have lifted my spirits during the most difficult times and celebrated with me in moments of joy.

I would like to express my sincere appreciation to the faculty and staff at the Technical University of Munich, whose knowledge and guidance have enriched my academic journey and broadened my understanding.

Thank you to the Chair of Connected Mobility, especially Prof. Dr.-Ing. Jörg Ott, for providing me with the opportunity to work on this meaningful project.

My deepest gratitude goes to my supervisors, Giovanni Bartolomeo and Dr. Nitinder Mohan, for their invaluable guidance, patience, and support throughout this work. Your mentorship has been essential to its completion.

Thank you.

Munich, July 2025

Abstract

Load balancers constitute critical infrastructure components within orchestrated frameworks, facilitating client-application connectivity. Traditional load-balancing techniques predominantly employ static algorithms that operate independently of real-time application and node state information. When deployed in edge computing environments, characterized by heterogeneous and dynamic infrastructure, these conventional techniques demonstrate insufficient performance and reliability. To address these challenges, this work presents comprehensive design considerations and criteria for developing an adaptive load-balancing system in edge environments, introduces a novel monitoring architecture, that aggregates telemetry data from applications and hardware through vendor-neutral interfaces, and leverages this information to make informed routing decisions through telemetry data analysis. By implementing an extensible policy engine, we establish a framework for the definition and enforcement of telemetry-aware load-balancing policies. Our experimental evaluation within an edge orchestration framework validates the proposed system's effectiveness, demonstrating that telemetry-aware load balancing can be achieved across the entire platform with minor performance overhead.

Contents

Acknowledgments **iii**

Abstract **iv**

1. Introduction **1**

2. Background **3**

2.1. Observability . . . 3

2.1.1. Logs . . . 3

2.1.2. Traces . . . 3

2.1.3. Metrics . . . 4

2.1.4. Telemetry . . . 4

2.2. Monitoring Platforms . . . 4

2.2.1. Prometheus . . . 5

2.2.2. Datadog . . . 6

2.2.3. Jaeger . . . 6

2.2.4. OpenTelemetry . . . 6

2.3. Load-Balancing Systems . . . 9

2.3.1. Envoy Proxy . . . 10

2.3.2. HAProxy . . . 10

2.4. Orchestration Frameworks . . . 11

2.4.1. Kubernetes . . . 11

2.4.2. Oakestra . . . 12

2.5. Related Work . . . 16

3. System Design **18**

3.1. Non-functional Requirements . . . 18

3.2. Functional Requirements . . . 18

3.3. Monitoring System Architecture . . . 18

3.3.1. Monitoring Targets and Collection . . . 21

3.3.2. Data Transmission . . . 21

3.3.3. Data Processing and Analysis . . . 21

3.3.4. Reactive and Proactive Measures . . . 22

4. Implementation **24**
4.1. Overview . . . 24
4.2. Added Components . . . 26
4.2.1. Monitoring Agent . . . 26
4.2.2. Monitoring Manager . . . 29
4.2.3. Routing Manager . . . 37
4.3. Adapted Components . . . 38
4.3.1. System Manager . . . 39
4.3.2. Root Service Manager . . . 39
4.3.3. Cluster Service Manager . . . 40
4.3.4. Network Manager . . . 40
4.3.5. Node Engine . . . 40
4.4. New Load Balancing Algorithms . . . 41
4.4.1. FPS Load Balancing Algorithm . . . 41
4.4.2. Underutilized Load Balancing Algorithm . . . 42

5. Evaluation **44**
5.1. Setup . . . 44
5.2. Test Configuration . . . 45
5.2.1. Test 1: Node Scaling . . . 46
5.2.2. Test 2: Application Scaling . . . 46
5.2.3. Test 3: Transition Latency . . . 46
5.3. Results . . . 47
5.3.1. Test 1: Node Scaling . . . 48
5.3.2. Test 2: Application Scaling . . . 55
5.3.3. Test 3: Transition Latency . . . 63
5.3.4. Non-Functional Requirements Evaluation . . . 66

6. Conclusion **69**
6.1. Summary . . . 69
6.2. Limitations and Future Work . . . 69

A. Appendix **71**
A.1. Monitoring Agent Configuration . . . 71
A.1.1. Builder Manifest . . . 71
A.1.2. OpenTelemetry Collector Configuration File . . . 72
A.2. Enabling the containerd Prometheus Metrics Endpoint . . . 75
A.3. Example OpenTelemetry Metadata File . . . 75
A.4. Example Formula . . . 76

A.5. Internal Metric Referencing Guide 76
A.6. Monitoring Agent - Collected Metrics 78

B. Implementing a new Load-Balancing Algorithm 80
B.1. System Manager Changes 80
B.1.1. Change the SLA Schema 80
B.1.2. Change the service management handler 81
B.2. Root Service Manager Changes 81
B.2.1. Implement Service IP Management 81
B.2.2. Implement database wrapper functions 82
B.2.3. Integrate database wrapper functions in the abstracted database requests 82
B.2.4. Implement the network management logic 83
B.2.5. Map the new strategy to the balancing policy 84
B.2.6. Assign new addresses to the service instances 84
B.2.7. What about IPv4? 85
B.3. Monitoring Manager Changes 85
B.3.1. Implement the new evaluator 86
B.3.2. Add the new evaluator to the collection in the routing policy . . 86
B.4. Network Manager Changes 87
B.4.1. Add the new Service IP type to the Table Entry Cache 87

C. Example Instrumented Application 89

D. Example Deployment Descriptors 94
D.1. Application with Load-Aware Routing Utilization 94
D.2. Application with Custom Metrics 96

List of Figures 98

List of Tables 99

Bibliography 100

1. Introduction

Edge computing has fundamentally transformed the computing landscape by bringing applications and services closer to end users, reducing latency and improving responsiveness for latency-sensitive applications [56, 29, 59]. To ensure application availability and optimal performance in these distributed environments, load-balancers serve as critical intermediary components between applications replica sets and clients, distributing incoming traffic across available resources [51, 45]. However, traditional load balancing approaches in edge environments are often based on fixed, static logic, that lack awareness of application-specific characteristics and real-time performance metrics [47]. These conventional methods are unable to adapt to changing application workloads, leading to suboptimal resource utilization and potential performance bottlenecks. This approach becomes particularly problematic in edge computing environments, where nodes exhibit diverse computational capabilities and resource constraints, leading to unbalanced load distribution and suboptimal resource utilization.

The lack of application awareness in traditional load-balancing approaches is a major limitation, as it prevents the system from making informed decisions based on real-time performance metrics. Consequently, there is a compelling need for the integration of telemetry data collection and real-time monitoring capabilities into load-balancing processes at the edge, enabling application-aware and adaptive routing decisions, which consider current system and application state [27].

The main contribution of this thesis is the requirements engineering and development of a vendor-neutral, asynchronous monitoring platform, enabling intelligent load-balancing based on real-time, cross-layer telemetry data. Our implementation introduces a novel modular architecture built on OpenTelemetry standards, consisting of custom monitoring agents and managers, which collect, process and analyze telemetry of distributed edge nodes. The monitoring managers's interface-driven architecture enables seamless integration and extension of new monitoring policies and processing logic with minimal overhead. This design is validated through the implementation of two load-aware load-balancing algorithms that demonstrate the system's extensibility. The system facilitates vendor-neutrality and modularity while demonstrating efficient operation for resource-constrained edge environments. We integrate our monitoring solution with an existing edge orchestration framework, and validate that the integrated monitoring system achieves sub-10% performance overhead in high workload scenarios, while enabling significant improvements through telemetry-aware load-balancing compared to traditional load-balancing approaches.

We start by establishing the needed context for telemetry-driven routing in edge environments in Chapter 2. We introduce observability in Section 2.1, followed by the general architecture of monitoring platforms and provide exemplary implementations and standards in Section 2.2. We continue with an introduction on load-balancing systems and existing solutions in Section 2.3 and finish with an overview on two state-of-the-art orchestration frameworks in Section 2.4.

In Chapter 3 we present the functional and non-functional requirements for the system in Section 3.2 and Section 3.1. Based on the requirements, we present a general monitoring system architecture in Section 3.3.

In Chapter 4 we present the realization of the proposed monitoring platform. We start by introducing the final system overview and the different components in Section 4.1. Based on the newly introduced components, we present each components implementation details in Section 4.2. On top of the new components, we give an overview of the adjusted system components in Section 4.3. Finally, we present the newly implemented routing algorithms in Section 4.4.

In Chapter 5 we present the evaluation of the proposed monitoring solution. We start by giving and overview of the evaluation setup in Section 5.1, followed by the introduction of the evaluation scenarios in Section 5.2. The results of the evaluation are presented in Section 5.3.

Chapter 6 concludes the thesis by summarizing the work in Section 6.1, followed by a discussion of the limitations discovered during the evaluation and the future work in Section 6.2.

2. Background

In order to better understand the technicalities of the thesis, we provide a background on important keywords and systems. We start by explaining the term and building blocks of observability and move on to presenting different monitoring platforms, implementing observability. We continue by explaining the term of load-balancing and give two standalone load-balancer implementations. We finish this chapter by presenting two orchestration systems.

2.1. Observability

"Observability is a system property that defines the degree to which the system can generate actionable insights. It allows users to understand a system's state from these external outputs and take (corrective) action." [32]

The observable external outputs can range from low-level metrics such as CPU timing information, memory, disk, and network utilization all the way up to application metrics such as transactions per second for databases, or error rates in API calls. These outputs can be used to deduct actionable information on the system state.

In the following we describe the three categories of external outputs used for observing a computer system.

2.1.1. Logs

Logs are structured or unstructured, text-based records generated by applications, services and system components about events as they happen. They are timestamped, and oftentimes human-readable information about what happened in the system [11], such as errors, warnings or user actions and allow the developer to monitor, understand and audit system behavior.

2.1.2. Traces

Traces give insights on the journey of a single request or transaction in the monitored application. Each trace consists of one or many spans, which is a single operation or step in the request's lifecycle. By incorporating metadata, such as start and end time,

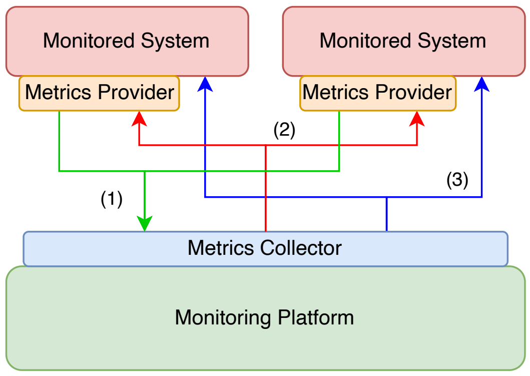

Figure 2.1.: General System Architecture of a Monitoring Platform

as well as the context, they allow developers to analyze the flow of a request through complex systems and identify possible bottlenecks, failures and performance issues.

2.1.3. Metrics

The OpenTelemetry Standard describes metrics as point-in-time measurements of services at runtime. They call the moment of capturing a measurement a **metric event**, which also include the timestamp of measurement and associated metadata [13].

Based on what information the metrics carry, they can give valuable insights into the availability and performance of a system or application, as well as indicate a trend or prediction for the system's future behavior.

2.1.4. Telemetry

Telemetry is the general term for the automated generation, propagation and collection of observable outputs of a system or application. Telemetry data consists of logs, traces and metrics. It is the enabling foundation of observability, as the collected telemetry is sent to a monitoring backend, analyzing, evaluating and visualizing the data. Based on the evaluations insights, the system can be optimized, improved and scaled.

2.2. Monitoring Platforms

There exist many different kinds of monitoring platforms, with each having their advantages and disadvantages based on their own specific use case. Figure 2.1 visualizes the general structure of well-established monitoring platforms, with arrows visualizing the different means of propagating the metrics to the collector.

1. The green arrow (1) visualizes the push-model for metric propagation. An agent connected to the monitoring platform regularly streams the updated metrics to the collector.
2. The red arrow (2) describes the pull-model. The collector regularly polls the agents, that are known to the collector, requesting the latest metrics.
3. The blue arrow (3) describes an agent-less solution, where the system provides an endpoint with the latest metrics. In combination with a pull-based monitoring backend, this solution is strictly limited to the monitored system's built-in metric capabilities, and therefore rarely utilized, as the scope of the proposed metrics is oftentimes way too limited, not providing enough context to give the "big picture". As an example, the built-in metrics endpoint of `containerd` [1] publishes metrics about its running containers at a configured endpoint on the host machine. However, these metrics alone, while giving some insight on the performance of the monitored containers, cannot be meaningfully interpreted, without any information on the host system performance metrics to put everything in relation to the total system resources available.

In the following sections we will present a selection of open-source, as well as enterprise monitoring solutions. We will give a short summary about its system architecture and put emphasis on the types of telemetry data each monitoring platform supports.

2.2.1. Prometheus

Prometheus [22] is a system monitoring and alerting platform originally established at Soundcloud in 2012. It is now a standalone open-source project, and maintained by a large developer and user community. To emphasize on the open-source nature of the project, it joined the Cloud Native Computing Foundation (CNCF) in 2016.
For its general system architecture, Prometheus is a pull-based monitoring solution, with agents deployed on the monitored nodes, which are responsible for the aggregation of system telemetry data. Push-based telemetry collection is also supported using an additional, intermediary Pushgateway [23] component.
Prometheus is designed for the efficient collection and storage of metrics. It does not support logs or traces, and uses a multi-dimensional data model for time series data, which are timestamped streams of unique metrics, identified by a metric name with optional key-value pairs called labels [19]. It is the main driver in the development of the OpenMetrics specification [21], which is a standard for the exchange of metrics between monitoring systems.

2.2.2. Datadog

Datadog [41] is a enterprise monitoring solution, developed by Datadog, Inc., offering a Software-as-a-Service (SaaS) monitoring platform for a wide range of system architectures, with great focus on Cloud infrastructure, like Google Cloud Platform (GCP) and Amazon Web Services (AWS). Its monitoring backend, which is responsible for the collection, storing and analysis of the telemetry data, is closed-source, however the code for its agent is available on GitHub [40]. The Datadog agent is push-based, and requires an API key to be configured in order to send metrics to the service provider's central Datadog backend. It provides support for the collection of logs, traces and metrics, allowing for a comprehensive view of the system's health and performance. The data format for metrics propagation between the agent and the monitoring backend is defined as Protocol Buffers [49], and also available on GitHub [39].

2.2.3. Jaeger

First released as open-source in 2016, Jaeger [6] is a telemetry platform originally developed by Uber Technologies and also in the hands of the CNCF today. It is designed to be a distributed tracing system, with a focus on performance and scalability.

It consists of a central collector, which is responsible for the aggregation, exporting of traces to a storage backend, as well as visualization of collected data. Agents, running on the monitored systems, are responsible for the collection of traces, which are created by the instrumented applications they host. Instrumentation is the process of adding code to the application, such that it supports the generation of telemetry data. The traces collected by the agents are then pushed to the collector.

The trace data representation adheres to the OpenTracing standard, which nowadays is superseded by the OpenTelemetry standard [18].

2.2.4. OpenTelemetry

OpenTelemetry [14] is a CNCF project, originally developed by the OpenTracing and OpenCensus projects. It is a set of APIs, libraries, agents and SDKs for the collection of telemetry data from distributed systems. It is designed to be a vendor-neutral standard for the collection of telemetry data.

It is not a monitoring platform, but a toolkit for the collection, processing and forwarding or exporting of telemetry data to a broad range of monitoring platforms, such as previously mentioned Prometheus, Jaeger and Datadog [16]. It allows for the instrumentation of applications in a standardized way, with support for the collection of logs, traces and metrics. Data propagation is done using the vendor-agnostic OpenTelemetry Protocol (OTLP), which is a protocol based on protocol buffers for the exchange of telemetry data between OpenTelemetry components. It supports both push and pull-based data propagation over HTTP or gRPC through the use of specialized receivers and exporters in the OpenTelemetry Collector.

OpenTelemetry Collector

The OpenTelemetry Collector [10] is a modular, compiled Golang [5] binary, that consists of a set of receivers, processors and exporters, building a pipeline for the aggregation, processing and forwarding of telemetry data. Each collector is configured at startup through a configuration file, which is used to define the pipeline. An OpenTelemetry Collector can also be configured to process multiple pipelines, each having their own set of receivers, processors and exporters.

The main goals of the OpenTelemetry Collector are:

- **Vendor-neutrality**: The OpenTelemetry Collector is vendor-neutral, and can be used with any monitoring backend, such as Prometheus, Datadog, Jaeger, etc.
- **Modularity**: It is modular, and can be extended with custom receivers, processors and exporters.
- **Performance**: Lightweight and performant under changing load and configuration.
- **Extensibility**: It is extensible, without needing to modify the core code.
- **Unification**: Using a single codebase, it allows to act as an agent or a collector for telemetry data.

The receivers are responsible for the ingress of telemetry data from the monitored systems. This can be achieved by either receiving the telemetry from an external entity, such as an instrumented application, or by generating the metrics themselves. With multiple configured receivers, metrics are received asynchronously, and are directly forwarded into the linear processing pipeline. A possible receiver set could be a Prometheus receiver, which pulls metrics from an endpoint of a Prometheus agent. Additionally, a receiver could be configured to scrape information about the Collector's host system, such as CPU, memory and disk usage. This could then allow for the monitoring of the Collector itself.

Processors are responsible for the processing of the telemetry data. They can be used to filter, enrich, aggregate or transform the data, before it is forwarded to the exporters. The OpenTelemetry Collector provides a set of built-in processors, such as the Batch processor, responsible for the aggregation of telemetry data from multiple

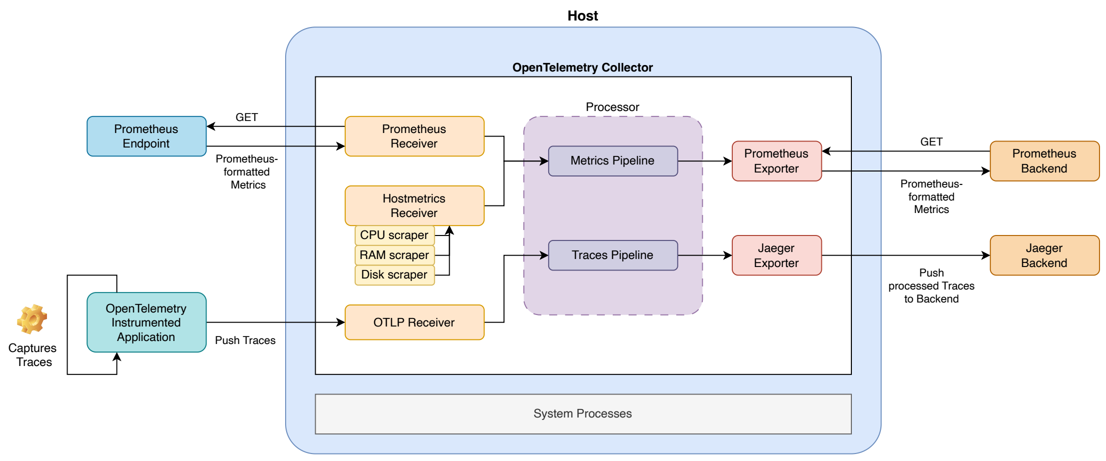

Figure 2.2.: Example of a OpenTelemetry Collector Pipeline

sources (receivers), and the Resource processor, responsible for the enrichment of the telemetry data with resource attributes, such as a hostname or a collector token for identification.

The exporters send telemetry data to the configured monitoring platforms or storage backends in their proprietary data formats and usually operate in a fan-out push-based manner. As an example, the OpenTelemetry Collector can be configured to forward metrics to a Prometheus server in the OpenMetrics format, or to a Datadog backend using the official Datadog Protobuf format. It could additionally export traces to a Jaeger backend through a separate, parallel processing pipeline.

Figure 2.2 gives an example for a pipeline of a OpenTelemetry Collector.

OpenTelemetry provides an official registry [16] with a collection of community developed Collector components, which can be used to extend the functionality of the OpenTelemetry Collector to any use case. Due to its vendor-neutral and lightweight design, it is possible to combine the OpenTelemetry Collector with any observability platform, in any environment.

OpenTelemetry SDK

The OpenTelemetry SDKs is the collection of libraries and APIs, used to instrument applications. They allow for the generation of telemetry data, and are available for a wide range of programming languages, such as Go, Java, Python, Node.js, and more [15]. It enables developers to achieve observability of their application, through its unified API and instrumentation libraries for logs, traces and metrics. Alternatively, OpenTelemetry provides zero-code instrumentation [17], which allows for the automatic

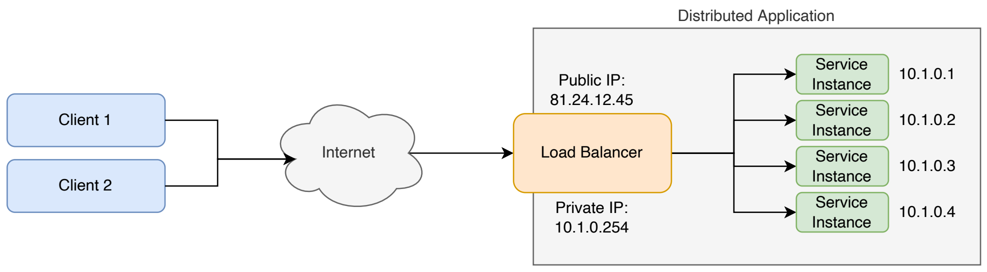

Figure 2.3.: Simplified Architecture of a Load-Balanced Service

instrumentation of applications without the need to modify the source code.

At runtime, the OpenTelemetry SDK captures the telemetry data, attaches contextual metadata, applies processing rules and forwards the telemetry data to the configured OpenTelemetry Collector.

2.3. Load-Balancing Systems

Load-Balancers are intermediary network components, responsible for the distribution of network traffic across an instance (replica) set of a service. Their functionality is often included in the context of a "reverse proxy", as they load-balance and proxy the request from the client to the service, and positioned between the client and the service.

Figure 2.3 gives a simplified view on the architecture of a load-balanced service. It is important to note that the load balancer is not part of the service, but a component that is deployed alongside it in order to provide the requested network functionality for the service. The load balancer exposes a single public endpoint to the client, which then proxies the request to one of the instances (replicas) of the service using its internal IP address, based on its load-balancing algorithm.

Load-balancers conduct their decision-making on two different levels of the network stack:

- **Layer 4**: The load balancer operates at the transport layer, where it makes forwarding decisions based on the TCP/UDP headers.
- **Layer 7**: The load balancer operates at the application layer, where it makes content-aware forwarding decisions based on the request headers, body or other content of the application request.

In addition to the decision-making based on static headers in the request, there exist load-balancer implementations that are able to make telemetry-aware forwarding decisions, based on observed metrics of the service instances, like Round-Trip-Time (RTT) or request error rates. In the following, we will present two such load-balancer implementations.

2.3.1. Envoy Proxy

Envoy Proxy [4] is a cloud-native, high-performance proxy, developed by Lyft, and now maintained by the CNCF. It is designed to be a universal edge and service proxy, and is used in a wide range of use cases, such as API gateways, service meshes, and service-to-service communication.

It is an application proxy, leveraging a dynamically programmable Layer 3 / Layer 4 filter chain architecture at its core, that perform different proxy tasks, such as load-balancing, routing, TLS termination, authentication, just to name a few.

An Envoy Proxy is deployed alongside the service on the host machine, and set up using listeners, clusters and routes. A listener, more commonly referred to as a frontend in the load balancing context, is a network endpoint, that receives requests from a client and forwards them to the appropriate cluster after applying the listener's configured filter chain. A cluster (in the load balancing context also referred to as a backend) is a load-balanced group of service endpoints, which share the same configuration, like load-balancing algorithm, health checks, and more. A route is a logical mapping of a request path to a cluster.

Envoy collects telemetry data in the form of internal metrics and traces about client requests to its clusters, which it uses in order to drive certain load-balancing policies [3]. The `active_requests` in its weighted least request load balancing algorithm is one such actively collected metric, which is used in the formula to determine the weights of the hosts in the cluster. Based on the resulting weights, the load-balancer then forwards the request to the appropriate host.

2.3.2. HAProxy

HAProxy is a renowned load-balancer and L4/L7 proxy, available as open-source [35], as well as commercial enterprise software developed by HAProxy Technologies LLC [50]. It is written in C, resulting in a high-performance, and low latency system, which is the result of its throughput-oriented design, where at the core it tries to optimize the utilization of the CPU cache by not inquiring higher-level cache lookups in the process, as well as sticking client connections to the same CPU core [37].

It is deployed on the host machine, and requires a configuration file to be provided at startup, which is used to configure the load-balancing policies and other proxy settings. Through the use of programmable Access Control Lists (ACLs), it enables the implementation of custom load-balancing policies, using a collection of internally collected variables and metrics [38].

At runtime, HAProxy allows for the reconfiguration using a runtime socket command line interface (CLI), which is used to dynamically update the configuration of the load-balancer. This is especially useful for the implementation of dynamic load-balancing policies, where the load-balancing priorities are updated based on the collected service telemetry data. Alternatively, the configuration can be updated using a RESTful data plane API, which is used to update the configuration of the load-balancer with no downtime, through seamless reloading.

HAProxy's rich feature set and high performance make it a popular choice in cloud orchestration platforms, such as Kubernetes, where it finds use as a load-balancer for service mesh implementations and enabled through the deployment of the HAProxy Ingress Controller [36].

2.4. Orchestration Frameworks

Orchestration frameworks are systems, that are responsible for the deployment, scaling and management of containerized applications in a distributed system. They abstract away the underlying infrastructure they operate on, offering a declarative model for the definition of computational, reachability and storage requirements of its hosted applications. Such frameworks ensure that the desired state is reached, by continuously monitoring the system and taking corrective actions to maintain the desired state. By encapsulating the complex operational tasks, like configuration propagation and fault recovery, they allow for developers to focus on application-level concerns, while providing operators with a centralized control plane for the management of the orchestrated clouds and edge environments.

In the following, we will present two such orchestration frameworks, one designed for cloud-native datacenter environments, and one for edge computing environments. In order to stay in the scope of this work, we will focus on the core networking capabilities and implementations of both frameworks.

2.4.1. Kubernetes

Kubernetes is an open-source container orchestration framework, originally developed by Google and now maintained by the CNCF.

At its core it consists of centralized control plane, which manages a set of nodes, that are responsible for hosting the containerized workloads. A set of nodes make up a cluster, and are all interconnected through a logical, shared cluster network. Containerized application instances, also referred to as Pods, are scheduled onto the cluster and assigned a cluster-unique Pod IP address, which is used to route the traffic to the Pod, enabling basic Pod-to-Pod communication across node boundaries. An application can consist of one or more replicas, which can be scheduled onto different nodes.

Additionally, Kubernetes provides a set of core networking primitives, such as Services and Ingresses, which are enabled by the `kube-proxy` [25] and used to further enhance the networking capabilities of the cluster.

Services

Services in Kubernetes are a network-level abstraction of a logical group of Pods. As Pods are ephemeral, and thus change their IP address over time on rescheduling, Services bridge this gap by providing a stable network endpoint to the application. There currently exist three different types of Services:

- **ClusterIP**: A cluster-unique IP address, which is used to route the traffic to an application. This is the default type of Service.
- **NodePort**: A NodePort binds a port on the hosting node's public facing interface. This is used to expose the application outside of the Kubernetes cluster at the node's public IP address and utilizes Network Address Translation (NAT) to route the traffic to the application. It is a superset of the ClusterIP type.
- **LoadBalancer**: Superset of the NodePort type, and instead of exposing the application at the node's public IP address, it is exposed at a static public IP address, which is assigned by the cloud provider.

Ingresses

Ingresses are a network-level abstraction of a logical group of Services. They are used to expose the application outside of the Kubernetes cluster, and are configured using a set of rules, that map a request path to a Service.

Ingresses are implemented using a reverse proxy, such as HAProxy, assigning an access path to the Service. Ingresses are typically managed by a dedicated Ingress Controller, which are notified about the changes to the Ingress definition and update the reverse proxy configuration accordingly.

2.4.2. Oakestra

Oakestra is a open-source, hierarchical orchestration framework, designed for, but not limited to, resource-constrained, heterogeneous edge computing environments,

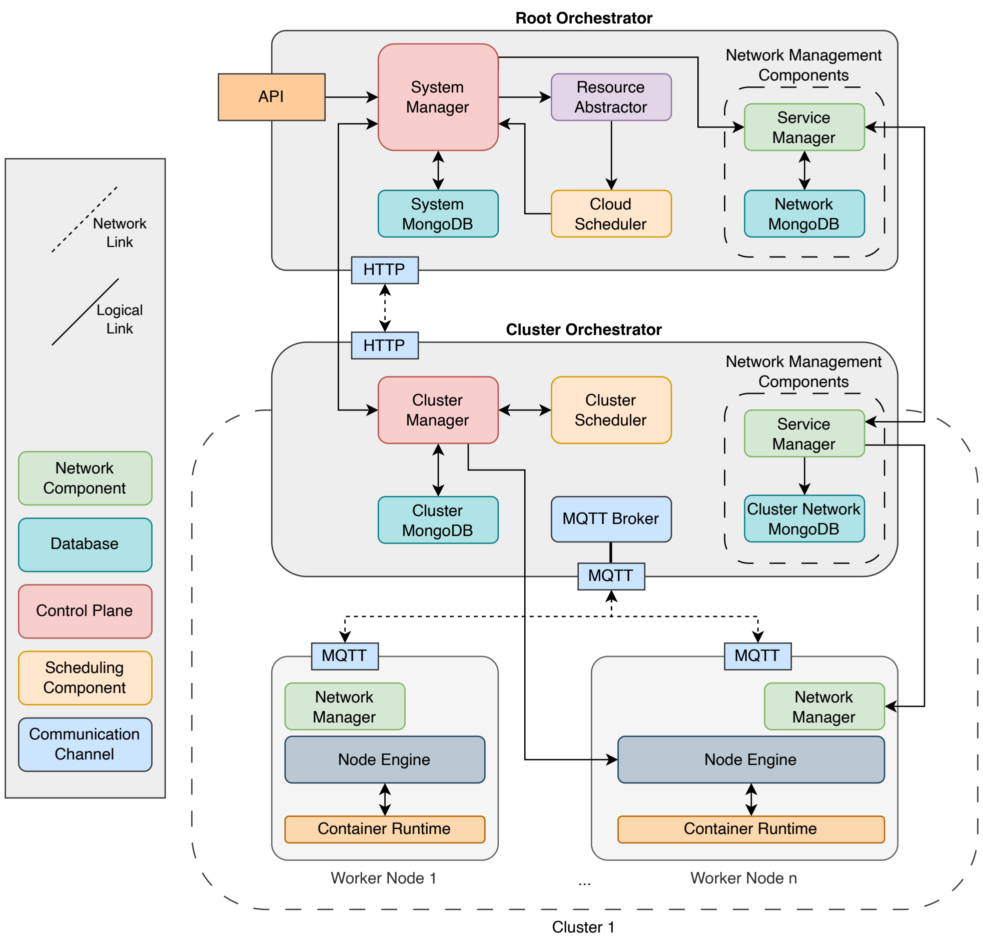

Figure 2.4.: Oakestra System Architecture

meaning it can be run on small edge devices, as well as large, monolithic datacenter environments.

It operates as a three-tier orchestration platform, consisting of a singular **Root Orchestrator**, which is the central control plane responsible for the management of the entire system and its deployments. Under the root lies a set of **Cluster Orchestrators**, which are responsible for the orchestration of the applications on its available **Worker Nodes**. Figure 2.4 visualizes the general architecture of the system.

In the following we give a brief overview of the core components at each tier, followed by a summary of the deployment process, with focus on the network related operations.

Root Orchestrator

The root orchestrator consists of multiple microservices, deployed as containers using Docker [42], with each microservice being responsible for a specific organizational task.

- **System Manager**: The system manager is the main entrypoint to the platform for developers, through its RESTful API. It delegates any developer initiated actions to the respective clusters for execution.
- **Cloud Scheduler**: Responsible for the scheduling of the applications to the available clusters, based on specific scheduling criteria defined by the developer.
- **Service Manager**: Management of platform network resources, such as Service IPs and Node Subnetworks.
- **Mongo Databases**: The root orchestrator hosts two different MongoDB instances for data persistence. One for general platform information, managed by the System Manager, and one associated with the Service Manager for general and application network management related information.
- **Resource Abstractor**: Standardizes the resource management by abstracting the underlying infrastructure information, such as the available resources on the worker nodes, into a common resource model.

Cluster Orchestrator

The cluster orchestrator can be seen as a structural twin to the root orchestrator with a different orchestration scope. It is responsible for the orchestration of the applications on its available worker nodes. It consists of the following components:

- **Cluster Manager**: Responsible for the management of the worker nodes and application deployments.
- **Cluster Scheduler**: The cluster scheduler calculates the optimal placement of a new application instance on the available worker nodes in its cluster.
- **Service Manager**: Delegates the reservation of network resources to the root orchestrator's service manager. Acts as an intermediary between the root orchestrator and the worker nodes for network related topics.
- **Mongo Databases**: Similar to the root orchestrator, the cluster orchestrator hosts two MongoDB instances for data persistence.

- **MQTT Broker**: Enables communication between the cluster orchestrator and the worker nodes. Oakestra specifically uses the Mosquitto MQTT broker [53], which supports the MQTT v5.0, v3.1.1 and v3.1 protocols.

Worker Nodes

Worker Nodes are the final components of the system, which are responsible for the actual execution of the applications. They are set up using two separate compiled Golang binaries.

The **Node Engine** is responsible for the deployment and management of the scheduled application instances on its available runtimes. The **Network Manager** enables the overlay network for the inter-application communication by keeping an internal proxy table, that is updated by the cluster's service manager. Both components communicate with the cluster orchestrator over MQTT.

Service IPs and Semantic Routing

Semantic Routing is a core concept of the Oakestra framework, which allows for the dynamic routing from and to applications across private network and cluster boundaries. It is implemented using a set of Service IPs, which are addresses taken from the private `10.0.0.0/8` and `fc00::/7` network space [52, 34]. They are statically assigned to application replica sets throughout their lifecycle, that are only available within the Oakestra overlay network.

Applications have multiple such Service IPs, with each adhering to a different semantic routing requirement. This allows for dynamic routing based on the clients' needs, such as the geographically closest application instance, or the instance with the lowest latency.

Resolution of Service IPs to application replica addresses is conducted by the network manager, who is regularly notified about changes in the application instance locations, as they are rescheduled to new worker nodes. An incoming request with a destination Service IP is first resolved to the application replica to forward the request to, before being proxied to the receiving host of the application.

Table Query and Interests

The resolution process from a Service IP address to application replica addresses is called **Table Query** and either happens synchronously during the proxying of a request, when no information about the application's Service IPs is available, or asynchronously on notification by the service manager because of a change in the platform network topology. The network manager requests the whole current application network information from the service manager, in order to then decide which application replica is contacted, based on the Service IP address and the semantic routing logic. At the time of writing, the network manager only supports the *Round Robin* semantic routing policy.

The triggering of a table query from a worker node inherits the declaration of a node's **interest** towards the application the Service IP is assigned to over a short period of time. This interest mechanism is used to keep the interested nodes updated about the resolution of Service IPs to application replica addresses. The interest is declared by the worker node, when it receives a request for a Service IP address, and is removed after a short period of time, when no more requests towards the application have been received.

2.5. Related Work

ViperProbe [48] implements an observability system, that analyses critical metrics of a microservice to be deployed through offline and online analysis, in order to determine the optimal working set of metrics to be collected. They implement the application-agnostic metric collection using the extended Berkeley Packet Filter (eBPF) [2], which is a technology that allows for the dynamic programmability of the Linux kernel. While limiting the scope of the metrics sent over the network reduces overall network load, it also reduces the amount of information available for the analysis of the application's performance and their possible causes for performance degradation. To counteract that, a lot of resources have to go into the analysis framework, in order to achieve a high level of confidence in the analysis results. On top of that, their black-box approach towards the analysis of the application's performance can miss important internal application signals, that are not immediately visible to the outside world. Therefore, we think that some degree of developer intervention is required to achieve a comprehensive view of the application's performance.

Further building on the black-box monitoring approach is the work of Brondolin et. al. [28], who also used eBPF in order to get a collection of different low-level performance counters of microservices deployed in a Kubernetes cluster. They implemented user-space monitoring agents, that regularly access the kernel-level metrics provided by eBPF, which are then pushed to a monitoring backend. While their solution has proven to be transparent towards the orchestrated application, performant and with low-overhead on the deployed infrastructure, their solution is strictly limited to the performance metrics collected by the eBPF programs.

Otero et. al. [55] propose a lightweight, inherently distributed middleware monitoring solution, which leverages the OpenTelemetry SDK and exporter in order to enable the collection of telemetry data, in this case traces, about APIs adhering to the OpenAPI specification [33]. They have shown that their solution is able enhance observability with minimal overhead and low system resource requirements.

In the load-balancing domain, Desmouceaux et. al. [30] propose 6LB, a load-balancing system that aims to leverage IPv6's Segment Routing (SR) [46] in order to achieve scalable and application-aware load-balancing. Therefore, they encode the load balancing decision directly into the packet header at network entry (ingress nodes), effectively steering traffic without the need to maintain state for each client connection. While incorporating application-layer information into the load-balancing decision, they do not base their decision making on telemetry information exposed by the application, but rather leave the final load-balancing decision to the responding servers, where each can express their availability based on their own load state.

Yao et. al. [60] propose Hybrid LB (HLB), which aims for optimizing the load-balancing decisions based on inferred server occupancy and processing speeds. By correlating the responsiveness of applications and the flow durations with resource exhaustion on the responding servers, they are able to make load-balancing decisions without the need for the active collection of telemetry. While this is great for the performance of the load-balancing system, the developer might still be interested to gain more insights into what is causing the resource exhaustion or slowdown of the application in the first place.

3. System Design

This chapter presents an approach to designing a telemetry-driven load-balancing solution for distributed edge-cloud orchestration platforms. We start by establishing a set of non-functional requirements, which provide the basis for the design process. We follow up with a set of concrete functional requirements, needed for the realization of the system. Finally, based on the established requirements, we present a high-level system architecture.

3.1. Non-functional Requirements

Based on the general goal, we can derive the following non-functional requirements, which are listed in Table 3.1 below. They are used to guide the design process and have to be generally applicable to all components of the system.

3.2. Functional Requirements

While the non-functional requirements are valuable to the system as a whole, we also need to define a set of functional requirements, in order to guide the design process towards a concrete solution that solves the research question of this work. Therefore, we provide our functional requirements in Table 3.2, which are specific to the monitoring backend. Based on these requirements, we can implement a specialized configuration of the monitoring backend, such that it implements the telemetry-driven load-balancing solution.

3.3. Monitoring System Architecture

In the following we show the components needed in order to realize a telemetry-driven load-balancing solution for an orchestration platform. Figure 3.1 visualizes the general composition of an orchestration platform with an example load-balanced application, before the adoption of a telemetry-driven load-balancing.

NFR1	*Modularity*	The system must be modular, allowing for the easy integration of new telemetry sources and propagation methods without deep architectural changes. Components must be able to be replaced or extended without the need to change the core functionality of the system, driving the maintainability of the system.
NFR2	*Interoperability*	The monitoring system must be able to be integrated with most already existing observability tools and standards for interoperability. This avoids vendor lock-in and allows for a broad range of possible use cases.
NFR3	*Compatibility*	While interoperability is important to avoid vendor lock-in, compatibility with different hardware architectures is just as important, allowing the monitoring system to be deployed on a wide range of heterogenous hardware.
NFR4	*Transparency*	The system must operate transparently to the orchestrated applications. It must not be a requirement for the developer to integrate his application with the monitoring system, and instead be offered as an optional feature.
NFR5	*Asynchronicity*	The system must be able to operate reliably at the edge, thus support asynchronous communication.
NFR6	*Scalability*	The system must be able to scale to the number of applications and replicas running on the orchestration platform without significant resource overhead or performance degradation.
NFR7	*Performance*	The system must be able to collect telemetry data from the orchestration platform in a timely manner, without significant overhead or delay.
NFR8	*Stability*	The system and its network must be able to handle the load of the orchestration platform, without significant performance degradation. In case of a network outage, the system must be able to safely recover from it.
NFR9	*Lightweight*	The system must be lightweight, requiring minimal resources to operate, in order to allow for the deployment on resource-constrained hardware.
NFR10	*Ease of Use*	The system must be easy to use and integrate into the orchestration platform.

Table 3.1.: Non-Functional Requirements Overview

FR1	The monitoring backend must be able to programmatically calculate and append custom metrics, based on the base metrics received from the monitoring agents and a declarative mathematical formula. It should allow developers to define their own metrics and formulas through a centralized interface.
FR2	The monitoring backend must provide a framework for the definition and implementation of custom policies. These policies must allow for the evaluation of internally programmed procedures, conducting their decision making based on the telemetry data.
FR3	The monitoring backend must allow for the definition of notification interfaces, that can be used to notify interested parties of the policy evaluation results. The notification interfaces must be able to be put into categories, allowing for the definition of different notification endpoints for different notification types.

Table 3.2.: Functional Requirements for the Monitoring Backend

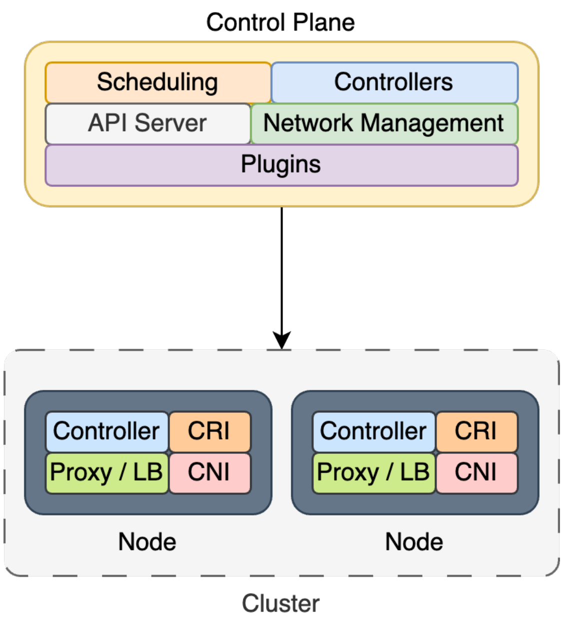

Figure 3.1.: General Orchestration Platform Architecture

When designing the system architecture, we wanted to answer the following questions, which were motivated by the definition of Telemetry as a process, as introduced in Section 2.1.4:

1. **What do we need to monitor?**
2. **How do we collect the telemetry data?**
3. **How do we process and analyze the telemetry data?**
4. **How do we use the processed data to make (load-balancing) decisions?**

In the following we will answer these questions in a general way, without going into the details of a possible implementation.

3.3.1. Monitoring Targets and Collection

For the data collection, it is important to define the entities you want to collect data from. In the context of application load-balancing, the most important entities are the distinct application replicas and the nodes they are running on.

The easiest and most common approach is to collect the telemetry data on the hosting node in a separate process. This allows to get an integral, cross-layer view of the application replica and the host state, and allows for the detection of possible correlations between the two.

3.3.2. Data Transmission

With the telemetry collected on the node, we need to define the transmission process to the monitoring backend. We already presented the two different approaches: pull and push based transmission. While the pull-based approach generally leads to a more efficient use of bandwidth, due to the fact that data is only transmitted when requested or needed, the push-based approach seems like the more natural choice for a telemetry-driven load-balancing system, as it allows for faster detection of changes in the state of the application replicas, due to the direct processing of the telemetry information as soon as it becomes available. Therefore, choosing an appropriate messaging protocol is crucial for the performance of the system.

3.3.3. Data Processing and Analysis

For the data processing we take motivation from the already existing monitoring systems, with a central monitoring backend acting as the data sink for the monitored nodes. As the monitoring backend has to be reachable by all monitored nodes, it seems natural to place the monitoring backend alongside the cluster's control plane components. The backend then processes and evaluates the telemetry data, allowing for a holistic view of the state of the system.

Given that we collect cross-layer metrics – metrics that are available on different layers of the system – we can conduct far-reaching analysis about the state of the system and its orchestrated applications. While insights about the host performance are important to detect first performance degradation, the conjunction with application- and network-level metrics allows for a more thorough root-cause analysis of any performance-related issues observed.

This aspect can ultimately be leveraged in order to implement specialized load-balancing algorithms, conducting thorough analysis on a per-application basis in order to steer traffic from unhealthy systems or instances towards better performing ones with respect to the load-balancing policy.

3.3.4. Reactive and Proactive Measures

Once the telemetry data is processed and analyzed by the monitoring backend, we expect a set of actionable items, which are either reactive or proactive measures. We interpret reactive measures as measures to be immediately taken, while proactive measures are corrective measures to be taken in the future, not critical to the operation of the system. Based on the type of the measure, the associated load-balancers (arrows with number 3 in Figure 3.2) are notified to take appropriate action. Therefore, an alarm and notification system is needed, which is responsible for the reconfiguration of the load-balancers, based on the actionable items, i.e. changes in the host and network state. Based on the type of the measure, the traffic is steered towards a backend instance, which, according to the associated load-balancing policy, is the better performing instance of the application replica set.

All proposed answers to the questions above result in the following simplified system architecture, given in Figure 3.2, with special emphasis on the components of the architecture imperative to the new design.

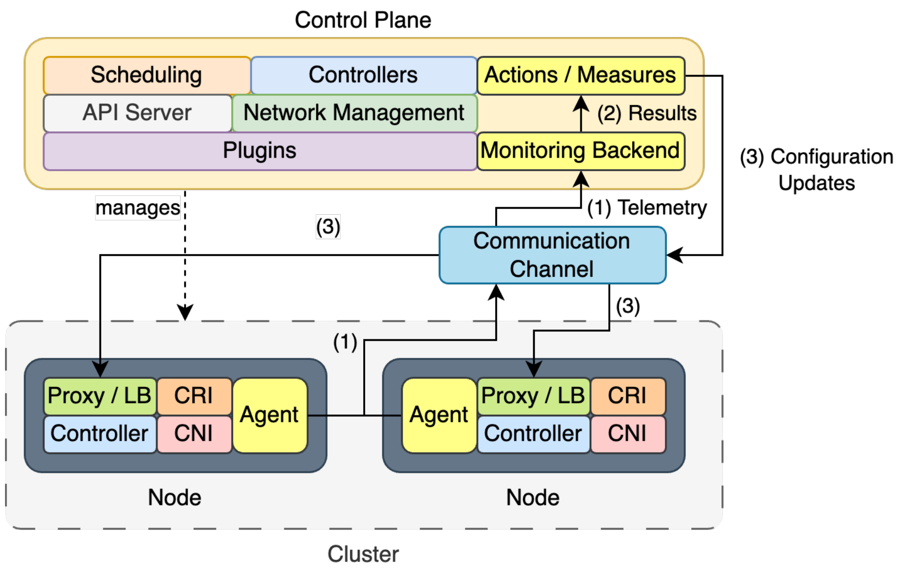

Figure 3.2.: General Orchestration Platform Architecture with Telemetry-Driven Load-Balancing

4. Implementation

This chapter presents the implementation of the system architecture described in Chapter 3. The proposed solution is built on top of the foundation of the Oakestra framework and its approach to semantic routing. This includes changes made to existing components of the framework, such as the worker's Network Manager and components of the control plane, such as the (Cluster-) Service Manager. We built the system on top of the current official main release of the Oakestra platform, which at the time of writing is version `v0.4.401`.

We start by giving an overview of the new system architecture with integrated monitoring, followed by giving the details of the added and adapted components, while explaining the choice of any frameworks, libraries and technologies used. Finally, we present the newly implemented telemetry-aware load-balancing algorithms.

4.1. Overview

This section provides a comprehensive overview of the extended Oakestra system architecture with integrated monitoring. Figure 4.1 visualizes the new architecture, highlighting the added components in yellow. Compared to the proposed architecture in Figure 3.2, the new architecture adds almost all given components, with the exception of the *Communication Channel* element.

Since the communication between the workers and cluster orchestrator was designed under the assumption that frequent disconnections are to be expected in the edge environment, this assumption still holds true in the propagation of the telemetry data, which flows from worker to cluster orchestrator. This was solved by using the existing MQTT broker for communication between the worker nodes and the cluster orchestrator components and inherently keeps messaging overhead low and reliable. This falls in line with our given requirements **NFR5** (Asynchronicity), **NFR6** (Scalability), **NFR7** (Performance), **NFR8** (Stability). More specifically, with the support for different Quality of Service levels in the MQTT protocol [54] and broker, the system aims to ensure that the telemetry data is delivered with the highest possible reliability, even in case of a network outage. We will elaborate on the client-side messaging assurances in Subsection 4.2.1.

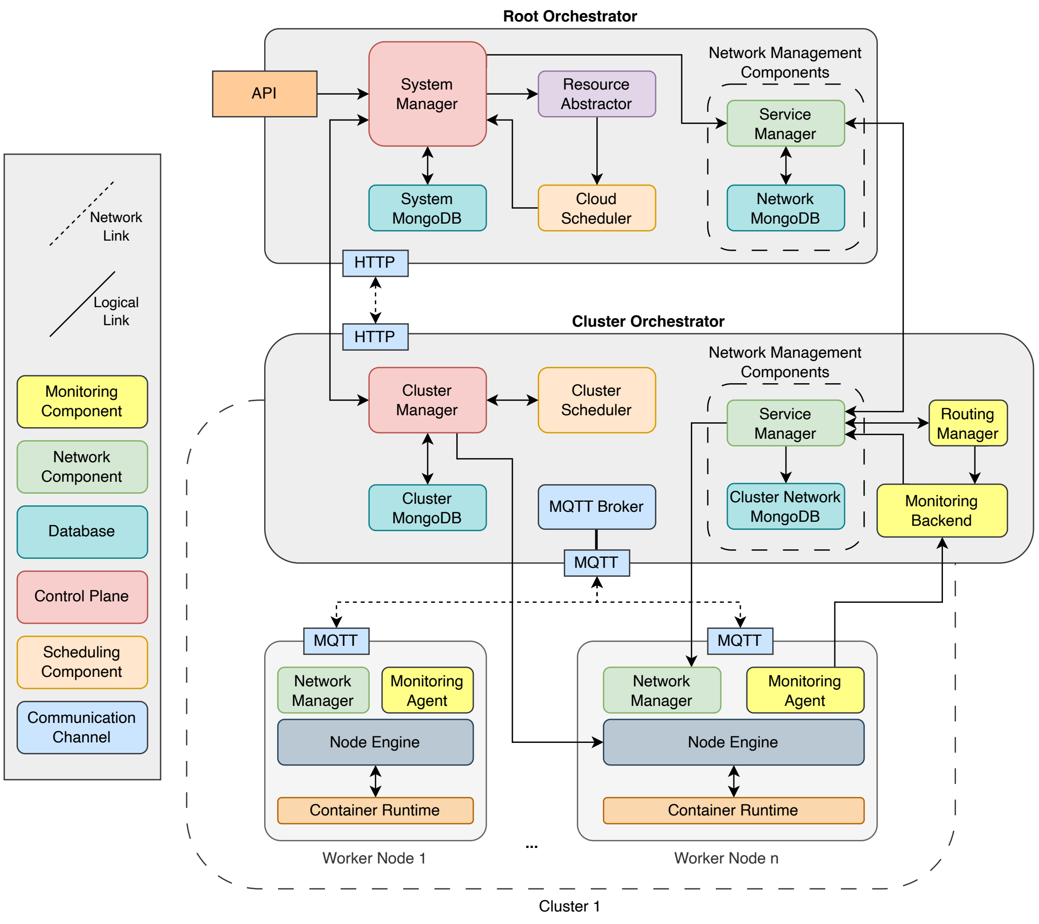

Figure 4.1.: General Oakestra System Architecture with Integrated Monitoring System

The type of telemetry data we collect and analyze in the system as of the publishing of this work is limited to metrics, as we wanted to keep the overall system design simple and focused on the main goal – enabling basic telemetry-driven network optimization. This however must not prohibit the extension of the system by adding support for the collection and analysis of other telemetry data types, such as logs and traces, as well as the addition of new propagation methods, as will be proven by the implementation details of the system.

4.2. Added Components

The following subsections go into detail about the newly added system components, focusing on the component design, implementation and their responsibilities. We structure this section in a bottom-up approach, starting with the monitoring agent in Subsection 4.2.1, followed by the monitoring manager in Subsection 4.2.2, and finally the routing manager in Subsection 4.2.3.

4.2.1. Monitoring Agent

In the design phase of the system we created, we determined that the agent should be able to allow for the collection of all types of telemetry data, metrics, logs and traces. As there exist many telemetry collection agent implementations in the observability ecosystem [24, 20, 40], most if not all of them have at least one major constraint: they are built for a specific platform or telemetry type, resulting in vendor lock-in, monetary costs, inflexibility or high maintenance overhead.

Our monitoring agent is realized as a custom OpenTelemetry Collector, compiled into a Golang binary, which runs as a standalone process alongside the other processes on the worker node. The build process is realized using the officially distributed OpenTelemetry Collector builder [8] and our own build manifest (A.1.1), containing the collector modules to be included in the binary. At startup, the compiled collector binary requires a configuration file (A.1.2), defining the collector pipeline and module configuration. Using the OpenTelemetry standard, we inherently benefit from its vendor neutral approach to observability.

The agent runs a custom pipeline, which is tailored to our monitoring use case. Figure 4.2 visualizes the pipeline modules and the remaining system components it interacts with. In the following subsections, we elaborate on the functionalities of each module in the pipeline. It is important to note that we chose to pin the version of the OpenTelemetry Collector, its modules, dependencies and build framework to version `v0.109.0`, as we observed frequent incompatibilities in the development cycle of our custom modules when updating the OpenTelemetry packages.

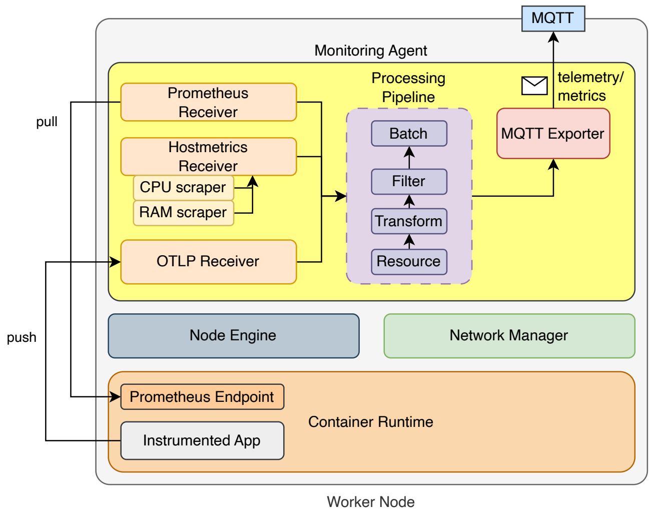

Figure 4.2.: Monitoring Agent Details

Module	**Responsibility**
`hostmetricsreceiver`	Collects metrics from the host system, such as CPU, memory, disk, network, etc.
`prometheusreceiver`	Collects metrics from the containerd Prometheus metrics endpoint.
`otlpreceiver`	Provides a generic interface for receiving telemetry of instrumented applications using the OpenTelemetry Protocol.
`filterprocessor`	Filters metrics based on a set of conditions.
`groupbyattrsprocessor`	Reorders and groups metrics based on a set of attributes.
`metricstransformprocessor`	Renames metrics into a more usable, human-readable format.
`resourceprocessor`	Adds the `machine` resource attribute to the metrics, more specifically the host identifier, as known to the cluster orchestrator.
`batchprocessor`	Batches metrics, sending them out in fixed intervals.
`mqttexporter`	Exports (publishes) metrics to a configured MQTT broker on a specific topic.

Table 4.1.: Monitoring Agent - OpenTelemetry Collector Pipeline Modules

Receivers

For the data collection, we determined three different sources: `hostmetricsreceiver`, `prometheusreceiver` and `otlpreceiver`. The `hostmetricsreceiver`, as the name suggests, collects metrics about the host system the agent runs on. Metrics collection from `containerd` deployed applications is handled by the `prometheusreceiver`, which queries the container runtime's Prometheus metrics endpoint. Note that this endpoint requires explicit activation in the `containerd` daemon configuration by the infrastructure provider, with configuration details provided in A.2. The `otlpreceiver` is the standard receiver for the OpenTelemetry Protocol, which is used to receive metrics from instrumented applications orchestrated by the platform.

A comprehensive list of the relevant metrics collected by the agent is given in Table A.6.

Processors

The processing pipeline performs some basic transformations on the collected metrics. It starts by adding a `machine` resource attribute to the metrics, which is used to identify the host the metrics originate from. This is done by the `resourceprocessor`. As the metrics collected by the `prometheusreceiver` are not in a format convenient for further processing, we use the `metricstransformprocessor` to rename the metrics into a more usable, human-readable format. Additionally, we use the `groupbyattrsprocessor` to group the metrics based on the labels set at the Prometheus endpoint. As one such label is the service name, this allows us to group the metrics by service, strictly separating the metrics of different services in their own metric resource packages. The `filterprocessor` is used to filter out metrics collected about containers that are not part of the Oakestra deployment namespace. Finally, the metrics are batched by the `batchprocessor` in order to aggregate the metrics into a single message.

Exporters

The metrics are exported to the monitoring manager by the `mqttexporter`, which is a custom exporter we implemented as part of this work in order to enable the publishing of metrics over MQTT. It acts as a regular MQTT client, connected to the cluster orchestrator's MQTT broker, publishing any data it receives from the processing pipeline to the `telemetry/metrics` topic.

As for the data encoding, we conducted a brief comparison of OpenTelemetry's JSON and Protobuf encodings, concluding that the Protobuf format is far more efficient, resulting in a smaller payload size. This stems from the fact that Protobuf is a binary format, which is more efficient to parse and encode than the JSON format, which is a text based data format, needing to encode every single character.

We summarize the responsibilities of each component in the pipeline in Table 4.1.

Design Evaluation

By deciding to use the OpenTelemetry as the framework for the monitoring agent, we can benefit from its well-established, highly modular architecture. This adheres to **NFR1** (Modularity), as changes to the agent can be done by simply adding, updating the versions, or removing components from the build manifest. Additionally, the agent compiles into a binary, which allows it to be packaged and distributed through version control systems, such as Git, and deployed on the worker node using fixed versioning. The OpenTelemetry standard is also well-established in the industry, offering a wide range of already implemented components for third-party monitoring tools, driving interoperability **NFR2**. This already shows by the use of the `prometheusreceiver`, which fetches metrics from the containerd Prometheus metrics endpoint. **NFR3** (Compatibility) is fulfilled, as Golang can be (cross-)compiled for different hardware architectures, allowing for the agent to be deployed on heterogeneous hardware, which also falls in line with Oakestra's vision. **NFR4** (Transparency) is given by design, as we *allow* for the developer to opt-in to further enhancing the application's monitoring by sending custom metrics to the agent's `otlpreceiver`. The monitoring is by no means a requirement for the application to be deployable on the platform. By sending the metrics asynchronously over MQTT, we enforce **NFR5** (Asynchronicity). In case the developer does decide to instrument their application, **NFR10** (Ease of Use) is provided through the standardized and well-documented OpenTelemetry SDK, allowing for easy adoption. Alternatively, they can also directly make use of the zero-code instrumentation provided by the official OpenTelemetry framework. **NFR6** (Scalability), **NFR7** (Performance), **NFR8** (Stability) and **NFR9** (Lightweight) will be shown in Chapter 5, the evaluation of the implemented system.

4.2.2. Monitoring Manager

The monitoring manager is the central component of our monitoring system, receiving metrics from the monitoring agents on the worker nodes, conducting analysis on the metrics it receives and relaying the results to the interested parties of the cluster. Given the vendor-neutral monitoring agent implementation, it allowed us to choose from a wide range of already existing backend solutions in the design phase. In order for us to be able to decide for the best solution, we compile and evaluate an extended list of requirements to the one already established in Section 3.2, before presenting our chosen framework and implementation details.

Evaluation of existing Observability Backends

Similar to the monitoring agent design phase, we evaluated a non-exhaustive collection of popular monitoring tools, such as Prometheus, Grafana, Datadog or Elastic. All considered solutions are very powerful tools, offering a wide range of features. A summary of our assessment is given in Table 4.2. In our evaluation we considered the following criteria, in addition to the previously established functional requirements:

- Vendor Neutrality: How easy is it to transition to a different observability backend?
- Open Source: Is the backend open-source, easy to extend and maintain?
- Licensing: Is the backend solution free to use, or does it incur a licensing fee?
- Telemetry Completeness: How complete is the telemetry support? Does it inherently support all types of telemetry data?

Platform	**Assessment Criteria**						
	Vendor Neutral	**Open Source**	**Licensing Costs**	**Completeness L / M / T**	**FR1**	**FR2**	**FR3**
New Relic	×	×	×	✓	✓	✓	✓
Datadog	×	×	×	✓	✓	✓	✓
Elastic Stack	×	×	–	✓	✓	✓	✓
Jaeger	✓	✓	✓	×	×	×	×
Prometheus	✓	✓	✓	–	✓	–	✓
OpenTelemetry	✓	✓	✓	✓	–	–	✓

✓= Excellent, – = Fair, ×= Poor

L/M/T = Logs/Metrics/Traces support

Table 4.2.: Comparative Analysis of Observability Backends

We based our final decision for the monitoring backend by prioritizing certain criteria over others. As this work was conducted in the scope of a research project with no funding, observability backends with licensing costs automatically fell out as options, thus eliminating the New Relic and Datadog commercial solutions as potential candidates.

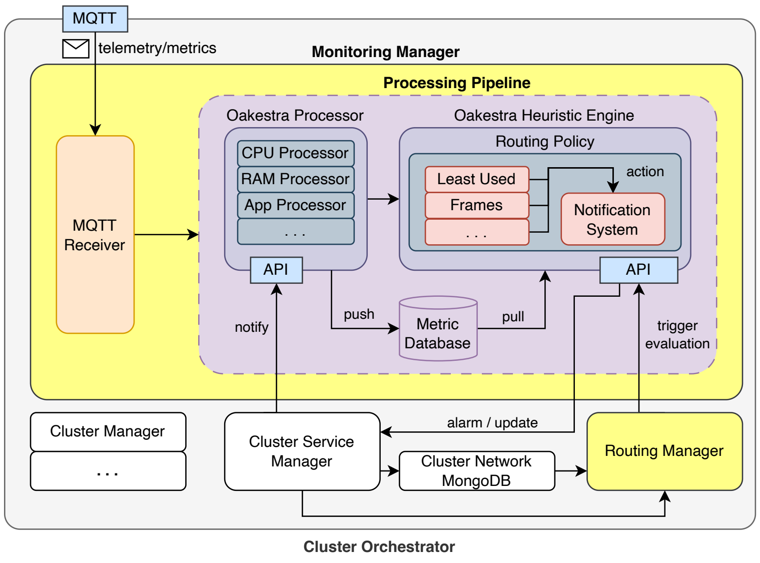

Figure 4.3.: Monitoring Manager Details

The Elastic Stack also fell out as a candidate, as it inherently requires too many resources [58, 31], which was expected, due to the many service-level dependencies it has, in order to provide its full functionality. This violates the system design requirements **NFR6** (Scalability) and **NFR7** (Performance), while also providing too many features out of the box, not imperative to the goal of this work.

Jaeger is a distributed tracing system, with no support for metrics. This makes the system not suitable for our expected use case, as we require at least metrics support with optimally support for logs and traces (Completeness requirement). The exact opposite is given for Prometheus, which provides great support for time-series metrics, but lacks support for logs and traces.

While not providing the full functionality we require out of the box, the OpenTelemetry framework provides a strong foundation for the implementation of a custom monitoring backend. It is vendor-neutral, open-source, has a strong community and is easy to functionally extend thanks to its modularity.

Implementation Details

Based on our evaluation, we decided to realize the monitoring manager as a custom OpenTelemetry Collector. While not a full-fledged observability backend as already pointed out in Section 2.2.4, we can greatly benefit from the OpenTelemetry standard, as already illustrated in Section 4.2.1.

For the collector internals, we implemented a collection of modules, whose implementation details, responsibilities and functional requirements are summarized in Table 4.3. Figure 4.3 visualizes the pipeline modules and the remaining system components it interacts with, which we will elaborate on in the following subsections.

Module	**Responsibility**
`mqttreceiver`	Receives metrics from a configured MQTT broker by being subscribed to a specific topic.
`oakestraprocessor`	Provides a programmable interface for the declarative calculation of custom service metrics, based on the base metrics received from the worker nodes.
`oakestraheuristicengine`	Provides a programmable interface for the definition of policies and their evaluation logic. Implements the audit and messaging logic of actionable outputs, based on the observed metrics.

Table 4.3.: Monitoring Manager - OpenTelemetry Collector Pipeline Modules

Receivers

The monitoring manager incorporates a single receiver, the `mqttreceiver`, which is technically identical to the monitoring agent's `mqttexporter`, with the small difference of subscribing to the `telemetry/metrics` topic instead of publishing to it, effectively receiving the metrics from the monitoring agents through the MQTT broker.

Processors

The `oakestraprocessor` is the first processor in the processing pipeline, who is responsible for the enrichment of the received metrics package using a set of subprocessors. Each subprocessor is responsible for the calculation of category specific metrics, such as CPU or RAM. We chose this design as a minor separation of concerns.

OpenTelemetry allows for the definition of a metric template in a `metadata.yaml` file, which defines, what metrics are generated by the pipeline module or submodule. An example for such a template is given in A.3, which describes for the CPU subprocessor, what resource attributes are expected to be set, along with metric-level attributes like state, as well as the metric name and unit. Using OpenTelemetry's `mdatagen` tool [12], we can generate the code scaffolding for the creation of OpenTelemetry metrics, which we use in the CPU and RAM subprocessors for enrich the telemetry data.

The CPU subprocessor is responsible for the calculation of service instance specific CPU utilization metrics, adding the metric `service.cpu.utilization` to the telemetry data. This metric contains two different states, `system` and `user`, for finer grained differentiation of the service CPU utilization.

The RAM subprocessor contains the logic for the calculation of service instance specific RAM utilization metrics, adding the `service.ram.utilization` metric. It calculates the RAM utilization for the following states: `used`, `slab_reclaimable` and `slab_unreclaimable`.

A special case in the subprocessor set is the application subprocessor, who does not contain any autogenerated code, as it is responsible for the calculation of metrics that are provided by the developer as part of the Service Level Agreement (SLA).

As for the implementation specifics of the metric calculation, we used a contract based approach. Each subprocessor maintains a list of calculation contracts, which are created and deleted with each application instance through a gRPC notification interface. On deployment, the cluster orchestrator notifies the `oakestraprocessor` about the deployment and any custom metric calculations from the SLA, which are then added to the contract list. The protocol buffer definitions for the notifications are published in our GitHub repository [66] and compiles into Golang and Python code, in order to sync the message type definitions between the cluster orchestrator and the monitoring manager. On top of the developer requested metric calculations, we implemented a set of default metrics, which we always calculate for each service instance, with the goal of being able to provide a basic set of load-balancing algorithms. We visualize the simplified contract workflow in Figure 4.4.

The contract datatype consists of a mathematical formula, the full service identifier it will conduct calculations for, an optional output state attribute and a list of input metrics and attributes it needs for the calculation. We therefore define a custom definition schema for the metrics, which can be used to declare variables inside the mathematical formula. An example for such a formula, as it is used in one of the subprocessors, is given in A.4, which calculates the current RAM utilization of a service. A thorough internal metric referencing guide is provided in Appendix A.5. Each subprocessor maintains a list of default contracts, which are statically defined in the code, and are always assigned for calculation on service instance deployment. This guarantees the basic functionality of the standard load-aware load-balancing algorithms. The only exception to this is the application subprocessor, who does not have any default contracts, but provides the main logic for the definition of custom metrics based on developer provided formulae, which are added to the contract list of the application subprocessor. It receives the custom definitions as part of the gRPC interface. An example for a custom deployment descriptor with custom metric definition is given in Appendix D.2. The contract calculated metrics are finally added to the telemetry data,

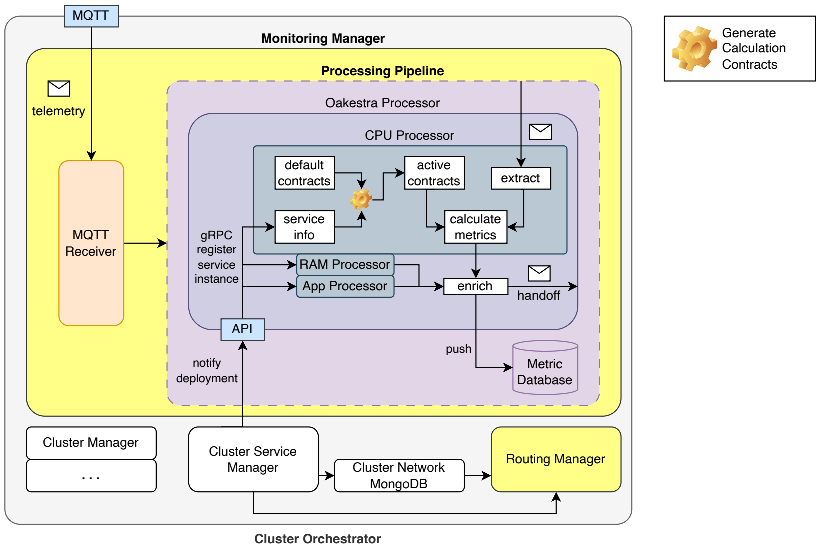

Figure 4.4.: Contract Interface Workflow in Oakestra Processor

which are used for the calculation of the load-aware load-balancing algorithms and also allow for the potential propagation by the exporters.

As the metrics are calculated on a per service basis, we need to be able to map the metrics to the correct service instances. By grouping the metrics in the collection stage per service, we can easily check the metrics resource attributes against our currently active contract list and only extract the metrics that are relevant for the calculation. We check for two conditions on the metrics resource attributes in order to identify and assign the metrics to the correct service instance "bucket":

- The `machine` resource attribute. This attribute is always present, and is used to identify the host the metrics originate from. This is set by the `resourceprocessor` in the monitoring agent.
- The `container_id` resource attribute. If it is present, the metrics part of this resource belong to a service instance. If it is not present, we generally assume that the metrics are host system related. This is set by the `groupbyattrsprocessor` in the monitoring agent.

The incoming and newly calculated metrics are then loaded into a basic, shared in-memory database, which stores the metrics according to our internal metric referencing definition and metric age, which effectively is a circular buffer with a pointer to the newest datapoint. This, in theory, allows for the calculation of metrics based on historical datapoints through the explicit definition of the datapoint age, which currently finds limited use in our implementation. We chose to go with our own in-memory database solution, instead of a third-party timeseries database solution, due to the data locality and greatly reduced latency of the metric retrieval. We will go into more detail on the data persistence topic in the evaluation chapter.

The `oakestraheuristicengine` is the second processor in the processing pipeline, responsible for the evaluation and auditing of policies it implements. It provides a framework of interfaces for the definition of policies, notification interfaces and evaluators. The framework aims to be as flexible as possible, in order to allow a multitude of different policies to be implemented. Each policy requires a unique identifier, a set of evaluators and a set of notification interfaces. The notification interfaces are the entities implementing the notification logic, with a predefined endpoint to which the notification is sent. A notification in more general terms is the actionable output of the policy evaluation that is sent to the interested parties. The evaluators are the final entities responsible for the analysis of the metrics based on its evaluation logic. They also have a unique identifier, which in combination with the policy identifier allows for the direct triggering of a specific evaluation routine using the HTTP endpoint the heuristic engine exposes. By externalizing the trigger mechanism, we allow for the interested parties to implement their own timing mechanisms for the evaluation of their interested policies. This way we can keep compute overhead low, as the evaluation of the policies is only triggered when needed / requested. An example for an evaluation request is given below, which is used to trigger the evaluation of the underutilized evaluator of the routing policy. The payload is a JSON object, which is the data expected by the specific policy in order to function.

```
POST /policy/routing/underutilized
Content-Type: application/json
{
  "appName": "benchmarking.test.application.service",
  "IpType": "underutilized"
}
```

At evaluation time, the policy fetches the current metrics relevant to the application from the in-memory database, which is updated by the `oakestraprocessor`, and passes them to the evaluator. The evaluator then calculates the scores for each service instance of the application, and returns them to the policy, along with the evaluator's notification conditions. The notification conditions are then evaluated against the results in order to determine whether a notification should be sent. If so, the policy passes the results to the notification interface, which forwards them to the interested parties.

Contrary to the `oakestraprocessor`, the `oakestraheuristicengine` does not generate any metrics for further propagation, but instead evaluates and creates actionable outputs based on the metrics the monitoring backend receives.

Using the interface framework we created, we implement an exemplar routing policy, which contains a set of evaluators, one for each routing algorithm. The routing policy contains two distinct notification interfaces, one for routing alarms, and one for routing notifications, with the receiving endpoints being the cluster service manager's new routing handlers. The former is used to notify the service manager about immediate issues, such as a service instance being completely overloaded, requiring immediate action to be taken, while the latter is used to notify about the current state of the application instances with respect to the routing policy without the need for a reaction. The evaluator identifiers are aligned with the orchestration platform's Service IP type naming. An evaluator takes a set of system and service metrics as input, which are fetched from the in-memory database at the moment of receiving the evaluation request. After evaluating all service instances, the scores are aggregated and pass through a simple notification evaluation algorithm, which again are specific to to the routing algorithm and implemented as part of the evaluator. This algorithm is used to determine whether the recent change in routing scores is significant enough to trigger a notification. If so, the scoring results are passed to the notification interface, which forwards the results to the interested parties.

Exporters

Our monitoring manager does not contain any exporters, as the generated metrics currently have no target external system. It is nevertheless possible, and even encouraged, to integrate the monitoring manager with a third-party monitoring backend, as for example the visualization of the metrics is currently not implemented in the monitoring manager.

Design Evaluation

Since the monitoring manager is also a custom OpenTelemetry Collector, a lot of points from the design evaluation of the monitoring agent in Section 4.2.1 apply here as well. **NFR1** (Modularity) is given by design, as the monitoring manager is a modular OpenTelemetry Collector. **NFR2** (Interoperability) is also fulfilled, because the monitoring manager can be integrated with third-party monitoring backends, as long as they are in line with the OpenTelemetry Collector interface. **NFR3** (Compatibility) is given due to Golang being a cross-platform language, allowing for cross-compilation and compatibility with different hardware architectures. **NFR4** (Transparency) is given,

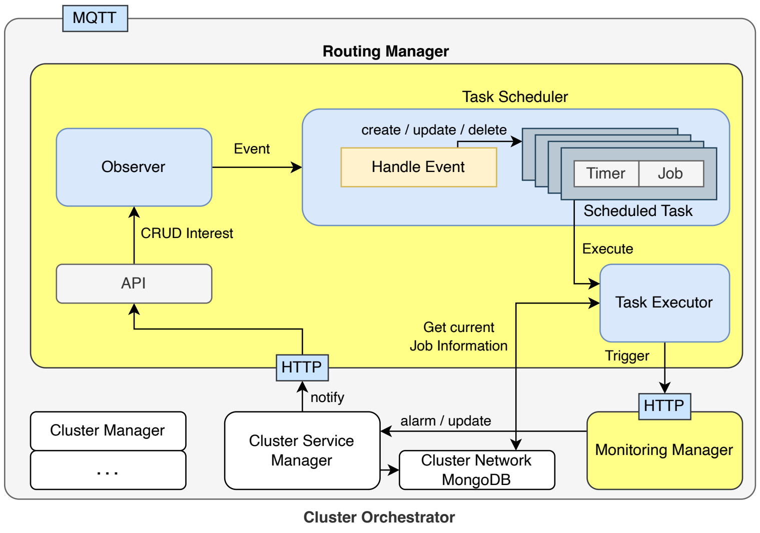

Figure 4.5.: Routing Manager Details

because the monitoring manager is not required in order for the developer to use the orchestration platform. To build on top of the already existing argument for **NFR1**, our modular approach for the processor internals allow for the flexibility of implementing new subprocessors, policies, evaluators or notification interfaces without the need for a major modification of the monitoring manager. As long as the new additions to the system are in line with the interface framework, they should be compatible and functional. This allows for the modular extension of the monitoring manager components, without the need to change already existing components. **NFR10** (Ease of Use) is assumed, as the interface-based implementation of the internal components abstracts away the complexity of the underlying implementation, allowing for the developer working on the monitoring manager to focus on the input and outputs of the components, without the need to understand the implementation details.

4.2.3. Routing Manager

The routing manager is a intermediary component between the monitoring and the networking components of the orchestration platform. It is written in Golang and responsible to regularly trigger the evaluation of the routing policies using the HTTP endpoint exposed by the `oakestraheuristicengine`. It is the main component responsible for the triggering of the routing policy evaluation on a per application basis.

Figure 4.5 visualizes the internals of the routing manager. At it's core, the routing manager is a simple HTTP server, with an event driven task scheduler. On interest registration, update and deletion, the service manager notifies the routing manager about the change through its API. This notification, which contains the application name, creates an internal event, which is propagated to a list of observers. Currently the only observer is a component called task scheduler. This component is responsible for the management of timed routing policy evaluation tasks. Based on the application name, the observer fetches the current application information from the service database, extracts the different Service IP address types set for the application, and spawns timed tasks for each Service IP address type for the given application. When a task is executed, the routing manager fetches the current service information from the service database, in order to provide the necessary metadata for the routing policy evaluation, like the current service instances deployed in the cluster. The task then queries the monitoring manager for the evaluation of the routing policy, before resetting the timer for the next evaluation.

For the routing manager, a subset of the system design requirements is given. **NFR1** (Modularity) is enforced, as the architecture allows for the extension of new observers for event handling, without the need to change the core functionality of the routing manager. **NFR2** (Interoperability) at this time is partially given, as the routing manager currently is tightly coupled with the service manager and his notification format. However, the implementation is not limited to it, given that the routing manager is modular, it is possible to extend the routing manager to support other notification interfaces for interoperability. By choosing Golang as the programming language for the routing manager, **NFR3** (Compatibility) is inherently given, due to the cross-compilation capabilities of the compiler. **NFR4** (Transparency) and **NFR10** (Ease of Use) are given for the same reasons as for the monitoring manager. **NFR5** (Asynchronicity) does not apply to the routing manager, due to its proximity to the monitoring manager on the control plane. Therefore, we do not need the same guarantees for the network communication paths as for the edge nodes. **NFR6** (Scalability), **NFR7** (Performance), **NFR8** (Stability) and **NFR9** (Lightweight) are left to be shown in the system evaluation in Chapter 5.

4.3. Adapted Components

In order to make the orchestration platform able to support the new telemetry-based routing, we had to make some adjustments to the existing components of the system. We provide a quick summary of the changes in the following subsections, with an in-depth description of the changes and new workflows in Appendix B.

We structure this section by starting with the system manager in Subsection 4.3.1, followed by the cluster service manager in Subsection 4.3.3, the network manager in

Subsection 4.3.4 and finally the node engine in Subsection 4.3.5.

4.3.1. System Manager

As the system manager is the entrypoint for the developer to the orchestration platform, we had to make minor adjustments to the schema for the deployment descriptors of new applications and their microservices. Therefore, we added new fields in the microservice descriptor, in order to support the reservation of addresses for our newly implemented balancing policies.

Additionally, we extended the microservice schema with a new `monitoring` field, which is used to provide a declarative description of the metrics to calculate in the `oakestraprocessor`. This, however, falls under the assumption that the developer is adhering to our internal metric referencing definition (as given in A.4), in order for the calculation to be possible, on top of making sure that the metrics are available in the pipeline.

Finally, we added another field to the SLA schema, namely `monitoring_class`, which triggers the specialized assignment of application specific Service IPs to the application. Currently, as a proof of concept, we have only implemented the `fps` monitoring class, which additionally sets the `fps` Service IP type. What this implies for the developer and the application is given in further detail in Section 4.4.1, where we talk about this specific new load-balancing mechanic.

4.3.2. Root Service Manager

The root service manager is the main network manager for the orchestration platform, responsible for the reservation of Service IPs for the applications. As we aim to make the implementation of new load-balancing algorithms as easy as possible for the developer, we had to abstract the general operations for the availability checking of the Service IPs based on the balancing policy.

Therefore we implemented a `IPAddressManager` who has an internal dictionary of available Service IP strategies and their respective address management logic. This way we can allow for the modular extension of the Service IP management logic on a per balancing policy basis.

Additionally, with the new `monitoring_class` field in the SLA schema, we implemented basic conditional setting of Service IPs based on the provided monitoring class. As the routing manager triggers the evaluation of the routing policy based on the Service IPs assigned to the application, we can hereby control, what evaluators are triggered for the application load-balancing.

4.3.3. Cluster Service Manager

As the receiving end to the monitoring manager's notification logic, the cluster service manager had to be extended to support the new alert and notification messaging, as well as the implied routing update logic. Therefore we implemented the two new endpoints in the service manager API, the backend logic to persist the new routing information to the database and implemented the alarm propagation to the concerned worker nodes.

Since the cluster service manager previously notified the network manager's through a specific MQTT topic – `nodes/<node-id>/update_available` – about any network related updates, we decided to reuse the already existing logic for the alarm handling. More specifically, upon receiving an alarm from the monitoring manager, it will first update the routing information in the database, before triggering the `update_available` notification to the worker nodes with an interest registered about the application that triggered the alarm. This notification causes the worker's network manager to conduct a table query to retrieve the new routing information.

While this approach is not the most optimal, due to the nature of the update mechanism and its multiple round trips, we can work under the assumption that this would not cause any disruptions of any in-transit traffic flows, whenever we update the routing in the network manager.

4.3.4. Network Manager

With the service manager notifying the network manager about the new routing updates, we needed to implement the update handling in the network manager, as well. While the workflow for the table query remains the same, the data structures behind the table queries have changed. Therefore, we extended the `TableEntry` object by the new routing table, containing the routing priority for each service instance based on the routing algorithm. With the new routing information being internally available, we also switched the decision making logic to a priority-based one.

We realized this by implementing a basic `Service IP Manager`, which is responsible for the selection of the respective service instance based on the routing algorithm. As Round Robin and Instance IP addresses are treated as special kinds of routing algorithms in the Oakestra network, we handle them as special cases inside the IP manager, as well.

4.3.5. Node Engine

The node engine had to undergo two minor changes in order for the monitoring agent to be able to transmit metrics to the monitoring manager under a common identifier. We implemented a minimal HTTP endpoint, which the monitoring agent uses to retrieve the cluster-assigned node identifier. This query is performed once on startup of the agent and internally used to add the `machine` resource attribute to the metrics. Furthermore, we extended the environment variables injected into the containerd container at deployment time, setting the `OTEL_EXPORTER_OTLP_ENDPOINT` environment variable to the host's IP address. This is needed in order to enable the automatic configuration of any providers initialized by the OpenTelemetry SDK in instrumented applications.

4.4. New Load Balancing Algorithms

In this section, we will present the new load-balancing algorithms implemented as part of this work. It is worth noting that these implementations are given as a proof of concept for the functionality of the system. Proper implementation, optimization, finer-grained control and tweaking of algorithm parameters at runtime are left as future work. We start with the image-pipeline focused `fps` load-balancing algorithm, followed by a more generally usable balancing algorithm, called `underutilized`, which focuses on the general utilization of available system resources. As a guideline for the implementation of the new load-balancing algorithms, we refer to the instructions given in Appendix B.

4.4.1. FPS Load Balancing Algorithm

The FPS load balancing algorithm is focused on the observable outputs of an image processing pipeline hosted in the orchestration platform. It is enabled by setting the `monitoring_class` to `fps` in the deployment definition of the application. Classifying the application in this monitoring context inherently means it provides a set of outputs, such that the application is able to be load-balanced based on its outputs. We leave the responsibility for the providing of the metrics to the developer, as they are application specific. To this end, we provide a small example of an instrumented application in Appendix C, which can be used as a reference for the developer to understand how to instrument an application for the monitoring platform, as it includes the specific configuration requirements.

For this load-balancing algorithm, we have reserved the Service IPv6 address subnetwork `fdff:4000::/21` as per the official documentation [26], which will be used to assign one Service IP for the semantic addressing of the application instances specific to this routing algorithm.

A table with the working set of metrics for the load balancing algorithm is given in Table 4.4.

Metric	State	Unit	Description
`service.fps`	`in`	`float`	The input frames per second of the application
`service.fps`	`out`	`float`	The output frames per second of the application

Table 4.4.: FPS Load Balancing Algorithm - Metrics Working Set

The general goal of this algorithm is to evaluate the current instance health based on image processing metrics exposed by the application. This could allow for the detection of internal queuing or buffering of the application, as well as the detection of a faulty instance, when fluctuations in the inputs/outputs are detected.

For our current proof of concept implementation, we calculate the exponential decay of the input framerate, in order to classify the instance as healthy or unhealthy. Based on the result, we get a value between 0 and 1, where 0 is the interpreted to be the lowest routing priority and 1 is the highest priority. We use the following formula in the evaluator to calculate the routing priority for the instance:

$$\text{priority}_{instance} = 1 - e^{-\lambda \cdot \text{service.fps.in}_{instance}} \tag{4.1}$$

Where λ is the decay rate, which is a constant that determines how quickly the result approaches 0 with respect to the input framerate. This has the effect, that if the input framerate approaches lower values, indicating lower performance or instance health, the priority approaches 0 as well. In our implementation, we fixed λ to 0.02 in order to get a very observable difference in our evaluation. Using this formula, we can guarantee an output priority between 0 and 1 for arbitrary positive values of the input framerate.

4.4.2. Underutilized Load Balancing Algorithm

The "underutilized" load balancing algorithm is a generally available load balancing algorithm, not tied to the setting of a specific monitoring class. It is enabled for all applications by default, and is based on the standard, not application-specific metrics provided by the monitoring agent. The Service IPv6 address subnetwork reserved for this algorithm is `fdff:3000::/21`.

It is focused on the resources the application instances are using, in order to conduct a load-balancing decision based on the most underutilized instance. Underutilized can be interpreted synonymously with "least-busy" or "most-idle".

In order to calculate the priority for each instance, we use the following simplified formula, which is the linear combination of the CPU and RAM utilization calculated by the `oakestraprocessor`, used in a exponential decay function, similar to the `fps` algorithm:

$$\text{priority}_{instance} = e^{-\lambda \cdot (\text{service.cpu.utilization}_{instance} + \text{service.ram.utilization}_{instance})} \tag{4.2}$$

Like for the `fps` algorithm, we use a decay rate of 0.02 for the exponential decay function. Given that the calculated utilization metrics cannot be negative, we can guarantee an output priority in the range of 0 to 1 for arbitrary values of the utilization metrics.

In order to illustrate the usage of the new load-balancing algorithms, we provide a small guide in Appendix D.1. It contains a full deployment descriptor of an application setup, which uses the `underutilized` load-balancing algorithm for load-aware routing.

5. Evaluation

This chapter aims to evaluate the performance impact induced by the new observation platform in terms of resource consumption and latency. Moreover we want to verify, whether our design choices fulfill the requirements set out in Chapter 3.

5.1. Setup

The testbed we conduct our experiments on is a collection of twelve virtual machines (VMs), running inside the computation cluster of the Rechnerbetriebsgruppe of the Technical University of Munich. Each virtual machine is provisioned with a virtualized Intel x86_64 4-core CPU @ 2.6GHz, 4GB of RAM and 80GB of disk space. All VMs are running Ubuntu 24.04.2 LTS on kernel version `6.8.0-60-generic`.

For the software requirements, we installed the `docker` engine using the official installation script, as given in the documentation [43], installing the `docker-compose` tool as well. In our testing environment, we used the `docker` engine version `28.2.2`, with `docker-compose` version `2.36.2`. The underlying container runtime is `containerd` version `1.7.27`. As an additional requirement, we installed the `golang` compiler version `1.24.4`.

All tests were conducted in a one cluster setup, where one VM node was chosen to run the root orchestrator, one VM running the cluster orchestrator and the remaining ten VMs deployed as worker nodes, depending on the requirements of the test, registered to the cluster orchestrator.

For the baseline measurements, we deployed the Oakestra platform based on the official documentation and their provided installation scripts, which pull the necessary microservice configurations and docker-compose files from the official Oakestra GitHub repository [7]. At the time of writing, the main release is version `v0.4.401`. For the observability enabled deployment, we extended the docker-compose files using overlay compose files with the necessary configuration. We published the observability enabled deployment files to forked GitHub repositories on the `main-monitoring` branch [63, 64]. In order for the deployment to work, it requires the checkout of the `routing-manager` [67] and `monitoring-manager` [65] repositories in the cluster orchestrator directory.

Parameter	Value
Demo-Application FPS Reporting Interval	1 second
Telemetry Collection Interval	1 second
Batching Interval	1 second
Routing Manager Task Timer	1 second
Monitoring Manager Alarm Threshold	Always trigger

Table 5.1.: Fixed Test Parameters

5.2. Test Configuration

In order to keep the amount of test cases and their parameters manageable, we decided to work with a fixed set of parameters for all subsequent test cases, which are listed in Table 5.1. In a production environment, these parameters should be adjusted at runtime in order to optimize the performance of the system and reduce the overall amount of computational and network overhead generated by the monitoring and load-balancing solution. The parameters have specifically been chosen so aggressively in order to show the performance and network overhead of the observation platform with enabled telemetry-driven load-balancing in a worst-case scenario.

We conduct three distinct tests, which we use in order to get a comprehensive overview about the impact on the overall resource consumption of the new observation platform. The first test is focused on the scaling of worker nodes and the inherent overhead on the monitoring manager resource consumption. The second test focuses on the scaling of the application instances, its inherent increase in telemetry data generated by the applications and the resulting increase in network and system load at the cluster level. The third test measures the routing update latency in the new telemetry-enabled load-balancing solution, which is quantified as the time between the moment the monitoring manager triggers a routing update and the moment it is reflected in the traffic steering of the network manager.

We implemented a scraper process for the measurements, running on each VM relevant to the respective test case. The scraper collects performance counters from the system, worker node processes and the docker containers, such as CPU, memory, disk and network usage, by reading the respective outputs in the `\proc` or `cgroup` directory and storing them in a CSV file. For the scraping interval we chose 1 second, which aligns with the telemetry collection interval of the applications. We conducted the measurements over multiple days, with each test case being run for 10 minutes, resulting in 600 data points per test case, which we binned into ten 60 second intervals for analysis. For the timing experiment, we adjusted the applications to write the timestamps about the distinct events to a file, which we then used to calculate the transition latency.

5.2.1. Test 1: Node Scaling

We start by examining the impact of the observation platform on the overall system load with a scaling number of worker nodes registered to the cluster orchestrator. Therefore, we compare the system load on the cluster orchestrator before and after the adoption of the observation platform, by examining the performance statistics in three different test configurations: one, five and ten worker nodes. The goal of this test is to get an estimation of the resource and networking overhead induced on a per worker node basis on the cluster, as well as the base resource consumption of the monitoring agent.

5.2.2. Test 2: Application Scaling

The second test scenario examines the overhead introduced by monitoring a scaling number of application instances, particularly focusing on the increased network and system load resulting from the greater volume of telemetry data generated per worker node and transmitted to the monitoring manager. For this test, we register a single worker node to the cluster and incrementally scale the number of application instances of a basic container writing image processing related information to standard output. Here again, we compare the base version against the telemetry-enabled version of the cluster orchestrator. Given that the monitoring platform is conducting the routing update analysis based on the configuration parameters given in 5.1, some degree of additional system load is to be expected. Our goal is to identify trends in resource consumption and network overhead of the monitoring agent as the number of application instances increases on a single worker node, as well as assessing the impact on overall system load at the cluster level.

5.2.3. Test 3: Transition Latency

Our third test case is focused on the transition latency of the network manager, more specifically the delay between the moment the monitoring manager observes a change in the state of the application triggering an alarm for a routing update and the moment the routing update is applied to the network manager, effectively steering traffic away from the affected application towards a better performing one.

We deploy a demonstrative instrumented image processing application [62], which we deploy with the `fps` monitoring class, reporting the input and output framerates of the application. It is worth noting that the container does not perform any actual work, it is only simulating the processing of images. By being in the `fps` monitoring class, it receives a `fps` Service IP address, resulting in the monitoring platform evaluating the application's performance based on the associated `fps` load-balancing algorithm. The instances report the framerate to the monitoring manager in a sine-wave pattern, with a period of 15 seconds. We deploy two application replicas with a deployment delay of approximately 7 seconds between them, thus creating an artificial route switching interval of at most 8 seconds between the two instances. The intersection between the two sine-waves can thus be interpreted as the transition point for the load-balancing decision, which is to be observed and handled accordingly by the monitoring manager, triggering a routing update towards the interested nodes. The demo-application exposes a basic HTTP server, with an endpoint returning the full application instance identifier.

In addition to the demo-application, we deploy a second application, acting as a consumer of the load-balanced demo-application. It specifically queries the now telemetry-aware `fps` Service IP address using `curl` [57], receiving the full application instance identifier from the responding instance.

By timing the moment the monitoring manager sends out the routing update through the alarm notification interface and the moment the consumer application observes the instance switch, we can measure the transition latency. In our testing, we conducted the measurements with two different querying frequencies, one with a frequency of 20 requests per second (every 50ms) and one with a frequency of 40 requests per second (every 25ms). We ran each test over multiple hours, acquiring a minimum of 3000 timed transition datapoints.

5.3. Results

In this section, we present the results of the three given test cases with a final evaluation of the performance-related non-functional requirements. While the root orchestrator plays a vital role in the overall system, we will not focus on it in this evaluation, as there were minimal changes to its implementation and full integration with the monitoring platform was not finished in time.

We start by presenting the results for the node scaling test case, followed by the application scaling test results. Finally, we present the results for the load-balancing transition latencies and finish with the evaluation of whether the results fulfill the non-functional requirements set out in Chapter 3.

5.3.1. Test 1: Node Scaling

CPU Usage

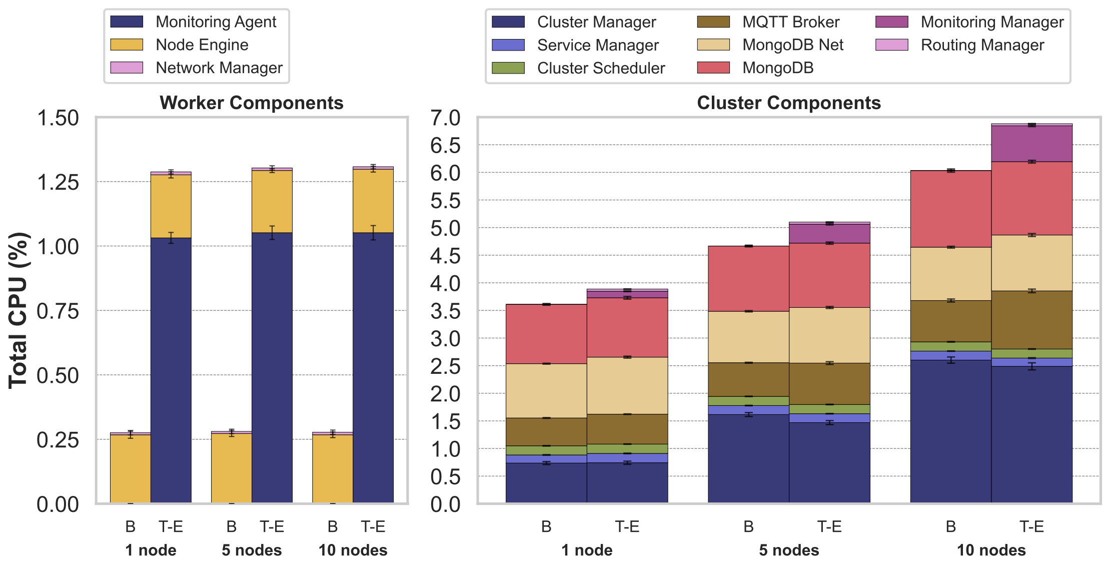

Figure 5.1.: Test 1: Node Scaling - CPU Usage

The average CPU usage of the different components in the platform is shown in Figure 5.1, with the worker node components on the left and the cluster orchestrator components on the right. We present the results in a side-by-side comparison between the baseline (B) and telemetry-enabled (T-E) version of the platform per test configuration.

Looking at the worker node results, we can observe a very noticeable increase (+410%) in CPU usage from the baseline 0.256% to 1.307% with enabled monitoring. This is due to the monitoring agent, which now runs on each worker node and regularly collects the telemetry data from the system and applications. For this test, this shows the baseline resource consumption of the monitoring agent with no applications running on the monitored worker. Across the different test configurations, the CPU usage does not noticeably fluctuate, as there is no correlation between the number of worker nodes in a cluster and the CPU usage of the monitoring agent or the singular worker node in general.

The cluster orchestrator also shows an increase in CPU usage due to the new monitoring and routing manager. The increase is however not as significant as on the worker node, compared to the baseline. The routing manager maintained an average CPU usage of 0.04% across the different test configurations, while the monitoring manager showed an average CPU usage of 0.123% for one active worker node, 0.343% for five active worker nodes and 0.650% for ten active worker nodes. With the enabled monitoring components, the total cluster orchestrator CPU usage increases by 4.3% relative to not running the monitoring services in the one node test configuration. For five nodes this relative increase is 8.07%, and for ten nodes 11.107%, which are almost exclusively attributed to the monitoring manager. The continuous increase in CPU usage in the monitoring manager is due to the increased amount of incoming telemetry data, which is propagated through the MQTT broker and processed and stored to the in-memory database by the monitoring manager. Even if no analysis on the telemetry data is conducted, the unmarshaling process of the telemetry data in the `mqttreceiver` is still significant.

Memory Usage

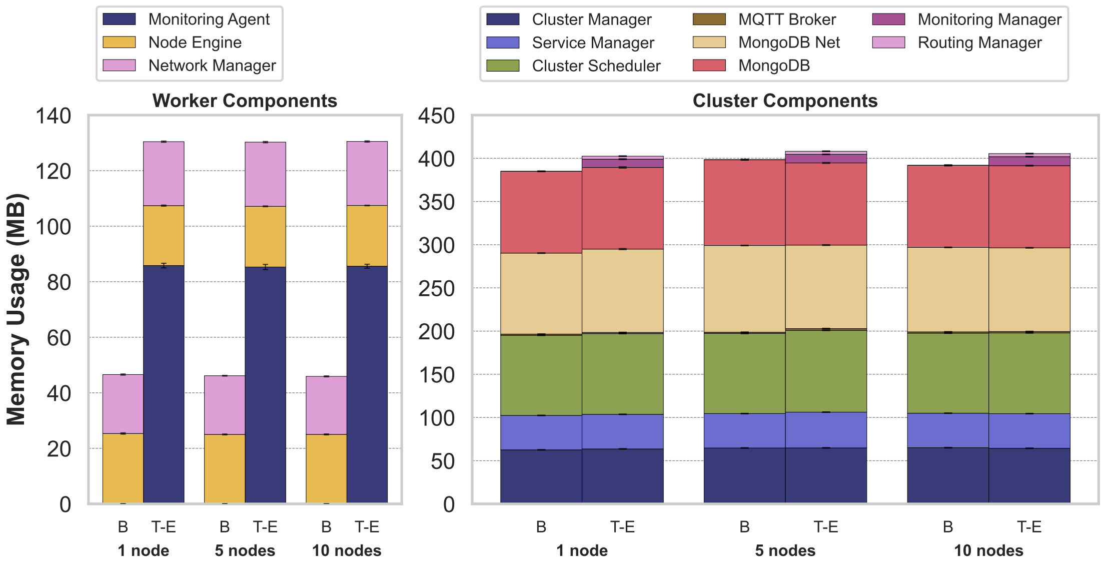

Figure 5.2.: Test 1: Node Scaling - Memory Usage

The memory usage of the different components in the platform is shown in Figure 5.2, following the same layout as for the previous CPU usage figure.

The overall worker node memory usage shows a noticeable increase of approximately 85MB (+190%) from the baseline of 46.18MB to 131.29MB. This, again, is due to the additional resources required by the monitoring agent. In order to get a better understanding of where the additional memory consumption is coming from, we took a closer look at the singular modules of the monitoring agent, which we use to collect the telemetry data. By excluding the modules from the build manifest, we can minimize the dependencies compiled into the binary, which allows us to get a more accurate picture of the memory consumption on a per module basis. The most significant increase in memory consumption is caused by the `prometheusreceiver`, a key module in the monitoring agent, with a total of 49.07MB of additional memory consumption. The remaining modules overall have not contributed as significantly to the overall memory consumption. As the `prometheusreceiver` implements the full feature-set for the telemetry collection of a Prometheus-based endpoint, a possible approach to reducing the memory footprint of that module is to implement a more lightweight, feature-limited HTTP client as a replacement.

For the cluster orchestrator, the monitoring manager required an average of 10MB of memory, with the routing manager requiring an average of 3.5MB of memory during operation. Compared to the baseline total memory usage, it results in a memory overhead of approximately 3.5%.

While the memory usage in the monitoring manager is expected to increase with the number of worker nodes, due to the inherent increase in telemetry data that is stored in memory, we could not observe a significant increase in our testing. With one worker node the average memory usage was 9.56MB, with five nodes 10.38MB and with ten nodes 9.93MB. So we cannot give a definitive answer by how much the memory usage increases with the number of worker nodes alone, as the amount of data collected is not significant enough for our observations.

Network Usage

In our network analysis we generally focused on the packet counters and bandwidth used. More specifically, we wanted to evaluate how much more traffic the accumulated telemetry data caused. The results are shown in Figure 5.3, which visualizes the average packets received per second by the worker node, cluster node and the cluster orchestrator components. The results for the bandwidth utilization are given in Figure 5.5. For transmission, we give Figure 5.4 for the packets sent per second and Figure 5.6 for the bandwidth utilization.

As we collected the data by looking at the kernel reported values for the network interface, background traffic is expected to cause irregularities in the results. We thoroughly examined the expected network traffic for the operation of the worker node and operation platform, which we will discuss briefly, before continuing with the analysis of the results. We conducted a analysis of the codebase of the worker node processes and came to the following conclusions, which will be important for the interpretation of the results throughout the rest of this evaluation.

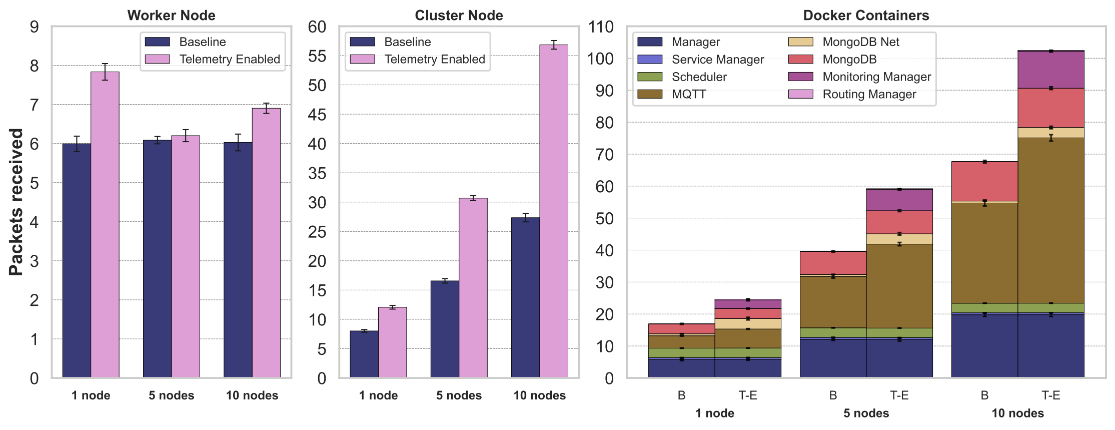

Figure 5.3.: Test 1: Node Scaling - Network Usage - Packets Received per Second

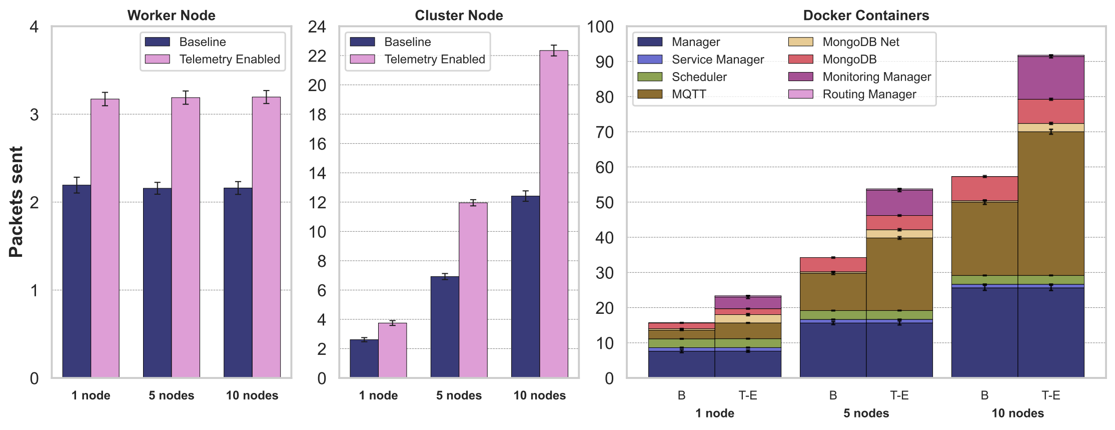

Figure 5.4.: Test 1: Node Scaling - Network Usage - Packets Sent per Second

- The worker node processes are exclusively talking over MQTT with the cluster orchestrator. This means that all traffic can be logged and analyzed on the cluster orchestrator's MQTT broker the processes are sending to.
- The node engine sends two messages to the cluster orchestrator in regular intervals. The first message is a service related message being sent to topic `nodes/<node-id>/jobs/resources`, and the second message is a system health related message being sent to topic `nodes/<node-id>/information`. With enabled telemetry, the monitoring agent additionally sends the telemetry data to the cluster on the topic `telemetry/metrics`.
- By analyzing the MQTT traffic at the broker, we can get an estimation of the expected packet count and bandwidth utilization for the worker node.

Based on those observations, we want to eliminate the amount of uncertainty in the results, i.e. be able to tell how much of the observed network traffic is to be attributed to the orchestration platform functions and **not** background traffic.

We analyzed the following traffic patterns for the different test configurations, which are summarized in Table 5.2. Given that the amount of data sent by the worker node scales with the number of services deployed on the worker node, we do not need to differentiate between the different test configurations, as the data sent by one worker node is the same for all test configurations of this test scenario.

Configuration	**Message**	**Size**	**Interval**	**Test**
N nodes, 0 services	`jobs/resources`	15B	2s	Baseline, Telemetry
	`information`	460B	2s	Baseline, Telemetry
	`telemetry/metrics`	3120B	1s	Telemetry

Table 5.2.: Worker Node Network Traffic Patterns

Based on the observations in Table 5.2, we can derive the expected packet count and bandwidth utilization for the worker node in the sending direction, which are reflected in Figure 5.4 and 5.6. Before the adoption of the telemetry sending, the worker node sent approximately 2.16 packets per second, with a bandwidth utilization of 0.42KB/s. These results capture most of the expected traffic, as the expected baseline traffic is 475B every 2 seconds (237.5B/s), transmitted as two distinct packets on the wire to the MQTT broker. With enabled telemetry, the worker node sends an average 3.19 packets per second, with a bandwidth utilization of 3.64KB/s. This falls in line with

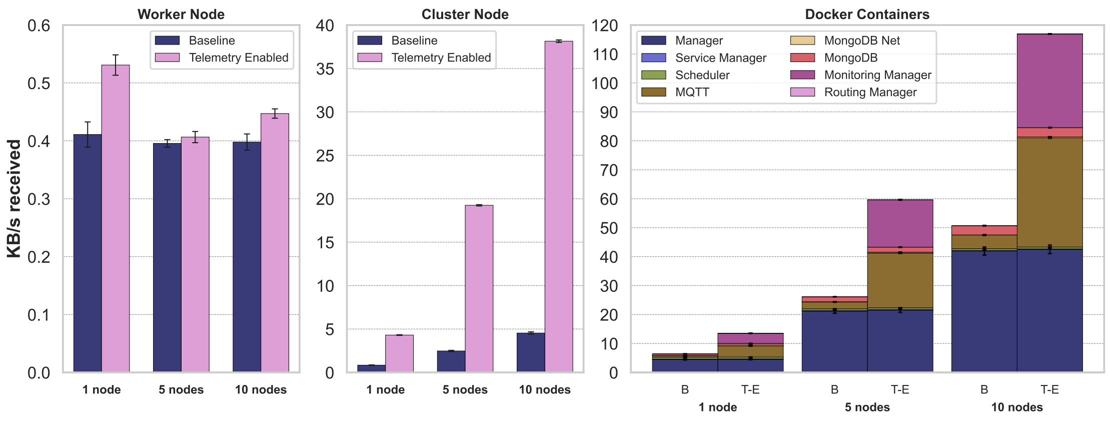

Figure 5.5.: Test 1: Node Scaling - Network Usage - KB/s Received

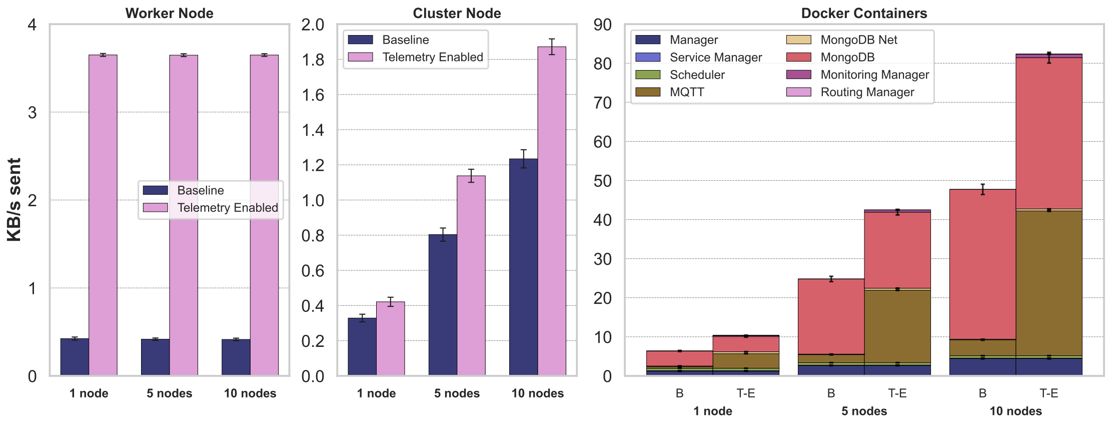

Figure 5.6.: Test 1: Node Scaling - Network Usage - KB/s Sent

Configuration	Direction	Scenario	Packets	KB/s	Growth KB/s
1 Node	**Send**	**Baseline**	2.61	0.33	
		Telemetry	3.74	0.421	28.03%
	Receive	**Baseline**	8.02	0.84	
		Telemetry	12.05	4.30	401.19%
5 Nodes	**Send**	**Baseline**	6.9	0.80	
		Telemetry	11.9	1.14	41.62%
	Receive	**Baseline**	16.5	2.47	
		Telemetry	30.6	19.24	678.09%
10 Nodes	**Send**	**Baseline**	12.4	1.23	
		Telemetry	22.3	1.87	51.63%
	Receive	**Baseline**	27.3	4.53	
		Telemetry	56.8	38.14	740.64%

Table 5.3.: Test 1: Node Scaling - Cluster Orchestrator Network Traffic

our network traffic analysis, where we expected an increase of 3120B/s. Nonetheless, it needs to be noted, that this is a significant increase in network traffic of +780% for the test case with no services deployed on the worker, with some degree of background traffic. For the receive statistics we have not been able to detect any significant changes between the baseline and telemetry enabled test scenarios.

On the cluster orchestrator side, we observe a close to linear increase in received network traffic. Therefore, we provide Table 5.3 with the summarized network traffic for the different test configurations and the respective growth rate. Generally, the received traffic is close to the amount of data sent by a worker node, multiplied by the number of worker nodes, which is expected.

The data sent by the cluster orchestrator also grows to some degree, but not as significantly as the received traffic. We suspect that most of the additional sent traffic are the ACK TCP packets sent by the MQTT broker to the worker nodes, as the amount of data sent remains rather small, with an average of 85 Bytes per packet (data sent divided by packets sent).

Breaking down the traffic observed by the components of the cluster orchestrator, we can observe an increase in the amount of data sent and received by the MQTT broker, which aligns with our previous observations. Due to the publish-subscribe (PUB/SUB) nature of the MQTT protocol, the observed data in the receive direction is expected to be very close to the data sent by the MQTT broker. Additionally, given that the telemetry data sent to the cluster is being forwarded to the monitoring manager, we can

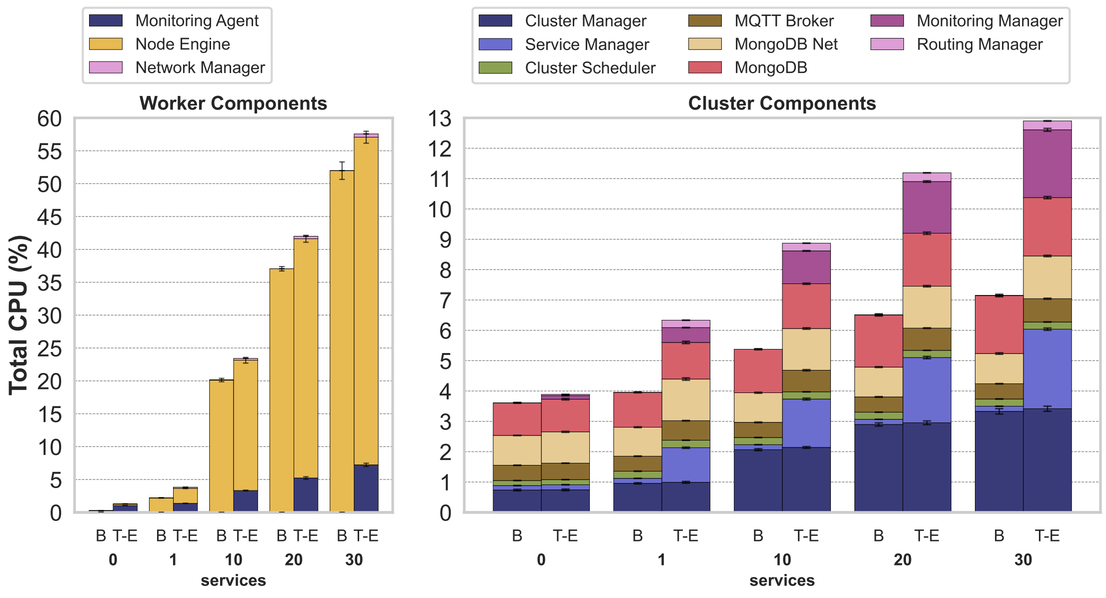

Figure 5.7.: Test 2: Application Scaling - CPU Usage

observe exactly this traffic pattern from the figures, where the surplus of data, w.r.t. the baseline, sent by the MQTT broker is received by the monitoring manager. The volume of data received by the monitoring manager is also close to the amount of overall data received by the cluster orchestrator in general, indicating that the telemetry data is the majority of observed network traffic across all test configurations.

5.3.2. Test 2: Application Scaling

CPU Usage

The CPU usage results for the application scaling test are given in Figure 5.7. Focusing on the worker node components first, we can observe a linear trend in overall CPU usage with the number of application instances. However, most of the CPU usage is attributed to the node engine, which, after thorough analysis of the source code, collects logs and system metrics from the worker node, as a basic feature even before the adoption of the monitoring agent. With more application instances it needs to scrape more data, scaling the CPU usage linearly. When looking at the numbers of the baseline 30 application instances result, the node engine accounts for 51.97% (99.97%) of the total 51.98% CPU usage. With telemetry enabled, the node engine's CPU usage equals 49.84%, which is equivalent to 86.59% of the total system load of 57.57% caused by the worker components. The monitoring agent also shows a linear trend in CPU usage, but not as significant as the node engine. In the 30 application instances test case, it accounted for 7.22% of the overall system load. Throughout the tests, the network manager remained very lightweight, with a maximum of 0.50% CPU usage in the 30 application instances, telemetry enabled test case.

For the cluster orchestrator components, we can also observe a similar load trend in the monitoring-, routing- and service manager. The scaling of application instances causes more calculations to be performed by the monitoring manager, which is reflected in the increase in CPU usage of 2.23% for 30 application instances. Due to the design of this test, where we always trigger an alarm for the routing update, the service manager needs to regularly perform the update of the routing table. Nevertheless, the observed CPU usage spike of the service manager going from 0.016% with no deployed application instances to to 1.13% with one single application instance is rather surprising. With 30 application instances, the CPU usage for the service manager is 2.62% of the total system load, which is not as significant as the previously mentioned increase. Therefore, we conducted a minor timing analysis of the routing update process, which has shown that the database operations take most of the time (approximately 3-7 milliseconds) during the routing update. Combined with the fact that the database access is a synchronous operation, it is highly likely that the service manager blocks for the duration of the database operation, which is why we observe the increase in CPU usage. This is the general access behavior across all components of the cluster orchestrator, which leaves room for performance improvements. With a CPU usage of 0.29% of the total system load in the 30 application instances test case, the routing manager was not a significant contributor to the observed system load, with only 2.26% relative to the total.

Memory Usage

Similar to the CPU usage results, the memory usage of the worker node components show a similar trend, which is shown in Figure 5.8.

Due to aforementioned metric and log collection in the node engine, its memory usage grows from 23.72MB with one service instance to 34.84MB with 30 service instances in the telemetry enabled test case. The monitoring agent shows a similar, but more steep trend, with a average memory usage of 101.13MB with one service instance to 136.55MB with 30 service instances, resulting in a per service instance memory requirement of approximately 1.2MB.

On the cluster orchestrator, the monitoring manager and routing manager turned out to be very memory efficient compared to the remaining cluster components. The monitoring manager accounted for 12.17MB with one service instance, to up to 19.11MB

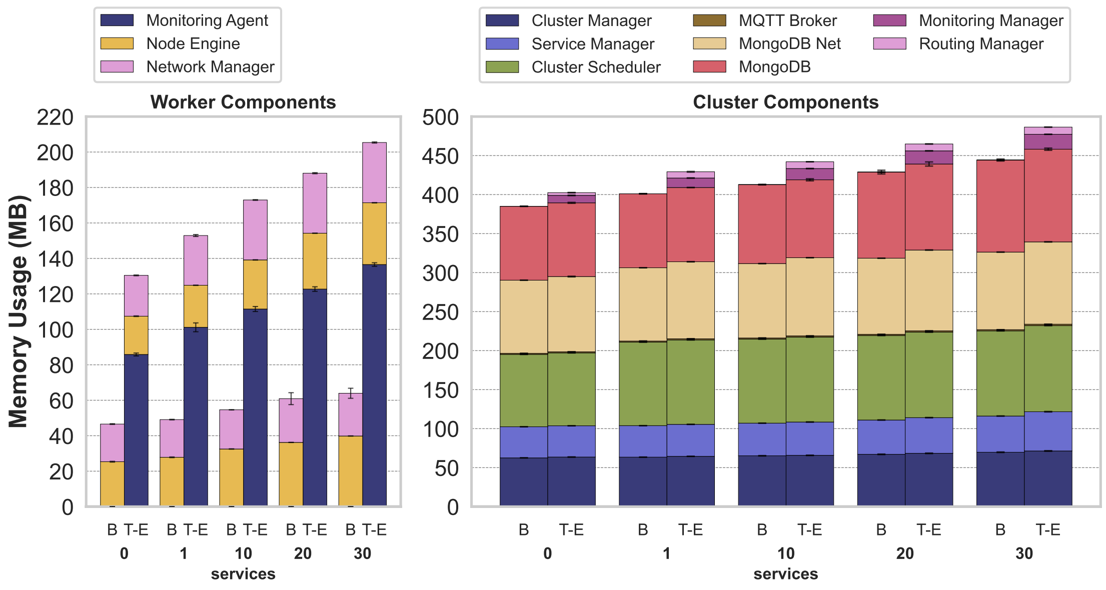

Figure 5.8.: Test 2: Application Scaling - Memory Usage

with 30 service instances, which is only 3.93% of the total cluster orchestrator memory usage. However, this results in a rough estimate of around 2.3MB per ten service instances, or 230KB per service instance, which needs to be taken into account when deployed in a production environment, where applications instances in a cluster can be in the thousands. The routing manager required 8MB with one service instance, scaling up to 9.21MB with 30 service instances, only attributing to 1.89% of the total memory consumption.

As a final addition to the memory usage results, it is worth noting that the observed growth in memory consumption of the monitoring manager is due to the implementation of the in-memory database, which keeps a snapshot of the last five observed data points per metric, per service instance. This was done in order to keep data locality and prevent the call to an external database. In a prototype implementation, we pushed the metrics to the MongoDB database, which however drastically increased the CPU consumption of the monitoring manager by up to 800%. A subsequent profiling analysis of the monitoring manager revealed that the encoding and decoding of the metrics to and from the database format took a significant amount of time, as well as the database operations. Therefore, we recommend to evaluate the use of optimized time-series databases, if data persistence is required to some degree, as the performance impact of the database operations can be significant. The time constraints of this thesis did not allow for a more thorough evaluation of this topic, which remains future work.

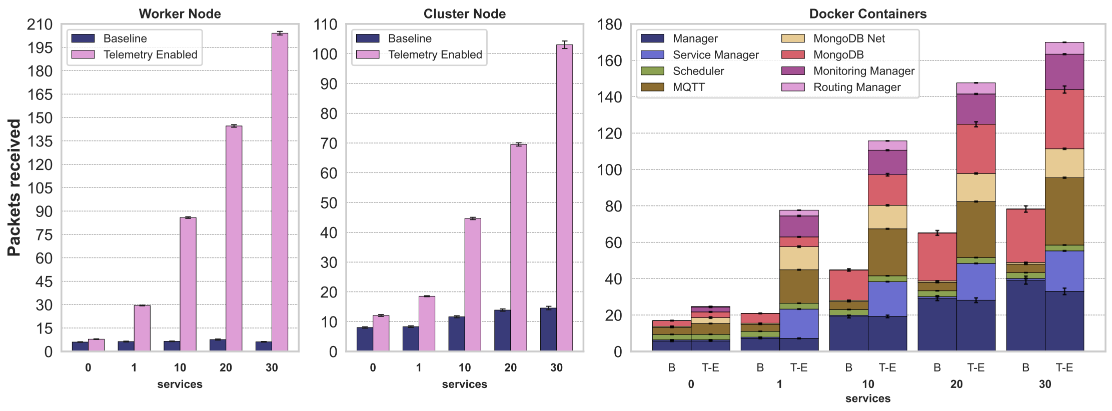

Figure 5.9.: Test 2: Application Scaling - Network Usage - Packets Received per Second

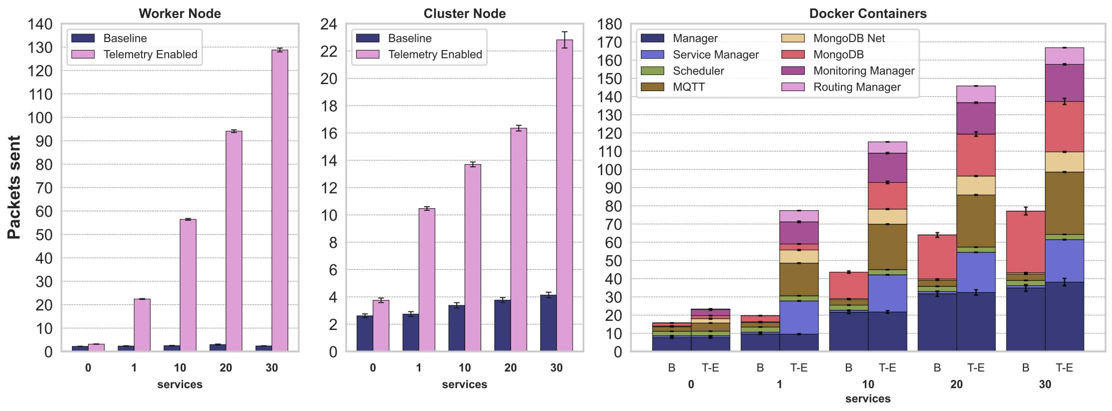

Figure 5.10.: Test 2: Application Scaling - Network Usage - Packets Sent per Second

Network Usage

The evaluation of the network usage with scaling application instances is the most impactful test, as it shows the costs of the telemetry data transmission on a per worker node basis. The results for the packet counts and bandwidth utilization in sending direction are given in Figure 5.10 and Figure 5.12, with the packet receive results in Figure 5.9 and respective bandwidth utilization in Figure 5.11. Following the same approach as for the node scaling test, we show the results for node, cluster and cluster orchestrator components.

Similar to the scaling behavior of the CPU and memory usage in this test, the increased amount of application instances, and thus increased amount of telemetry data also cause a very noticeable increase in network usage. Starting with the sending

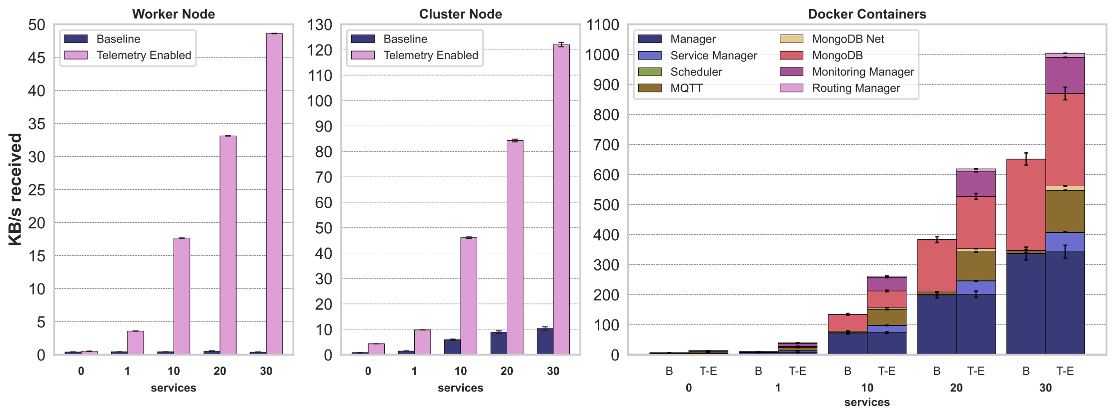

Figure 5.11.: Test 2: Application Scaling - Network Usage - KB/s Received

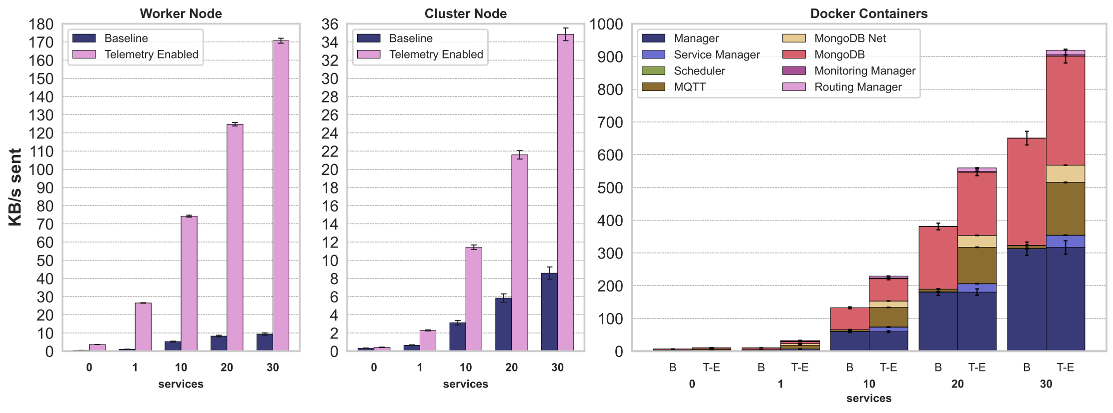

Figure 5.12.: Test 2: Application Scaling - Network Usage - KB/s Sent

direction, we can observe a very steep increase in the packet count and bandwidth utilization. On the worker node, the outgoing packet count increases by an average of 34 packets per ten instances, topping out at 128.75 packets per second with 30 instances. Compared against the baseline of 2.38 packets, this results in a percentage increase of +5309%, or 54 times more packets than the baseline. The bandwidth utilization increased from a baseline test with 30 application instances and an observed bandwidth of 9.4KB/s to 170.66KB/s, which is a +1715% increase.

With more data being sent by the worker nodes, the cluster node inherently receives more data as well. In our testing the cluster orchestrator received an average of 102.99 packets per seconds, and an associated bandwidth utilization of 121.97KB/s, resulting in an overall increase of +607.4% in packets received and a +1086.3% increase in bandwidth utilization for the test case with 30 application instances.

As already pointed out in the previous CPU test results, the cluster orchestrator components also show a very significant increase in network usage, especially the MQTT broker, service manager and routing manager. This is however expected and shows the current worst case scenario for the resource requirements of the monitoring platform.

In order to get a better understanding of what data is actually flowing in what quantity, we conducted a similar analysis of the MQTT broker traffic, this time however differentiating the traffic volume based on the amount of application instances deployed, which is summarized in Table 5.4. Additionally, now that the monitoring platform is actually conducting analysis on the telemetry data and triggering routing updates, we provide the additional traffic volumes generated from this workflow with specific sending annotations, where **W** denotes the worker node and **C** the cluster orchestrator. We also provide a visualization of the traffic patterns occurring every second in this test scenario in Figure 5.13.

When we apply the extracted data from Table 5.4 to the observed results, we can see that the telemetry data is once more the majority of the observed network traffic. The frequency of the routing update also weighs into the results, as it is triggered every second, resulting in the sending of the `update_available`, `tablequery/request` and `tablequery/result` messages. While the first two are not causing any significant increase in network traffic based on our observations, the latter scales with the number of application instances, resulting in a significant increase in network traffic of up to 21890 Bytes that flow from the cluster to the worker nodes, covering close to 2/3 of the total observed network traffic sent by the cluster node.

In further analysis of the `jobs/resources` message size, we noticed slight discrepancies between the expected and the observed data. While the expected message size for 30 application instances is 35442B every two seconds, so approximately 17700B/s, or 17.7KB/s, we only observe an average of 10.28KB/s being sent by the worker node

Configuration	Direction	Message	Size (KB)	Interval	Test
0 services	**W -> C**	`jobs/resources`	0.015	2s	B, T-E
	W -> C	`information`	0.46B	2s	B, T-E
	W -> C	`telemetry/metrics`	3.12B	1s	T-E
1 service	**W -> C**	`jobs/resources`	1.195	2s	B, T-E
	W -> C	`information`	0.46	2s	B, T-E
	W -> C	`telemetry/metrics`	7.5	1s	T-E
	C -> W	`update_available`	0.026	1s	T-E
	W -> C	`tablequery/request`	0.047	1s	T-E
	C -> W	`tablequery/result`	0.824	1s	T-E
10 services	**W -> C**	`jobs/resources`	11.814	2s	B, T-E
	W -> C	`information`	0.46	2s	B, T-E
	W -> C	`telemetry/metrics`	37.68	1s	T-E
	C -> W	`update_available`	0.026	1s	T-E
	W -> C	`tablequery/request`	0.047	1s	T-E
	C -> W	`tablequery/result`	7.3	1s	T-E
20 services	**W -> C**	`jobs/resources`	23.628	2s	B, T-E
	W -> C	`information`	0.46	2s	B, T-E
	W -> C	`telemetry/metrics`	71.34	1s	T-E
	C -> W	`update_available`	0.026	1s	T-E
	W -> C	`tablequery/request`	0.047	1s	T-E
	C -> W	`tablequery/result`	14.6	1s	T-E
30 services	**W -> C**	`jobs/resources`	35.442	2s	B, T-E
	W -> C	`information`	0.46	2s	B, T-E
	W -> C	`telemetry/metrics`	105.02	1s	T-E
	C -> W	`update_available`	0.026	1s	T-E
	W -> C	`tablequery/request`	0.047	1s	T-E
	C -> W	`tablequery/result`	21.89	1s	T-E

Table 5.4.: MQTT Broker Observed Traffic Patterns

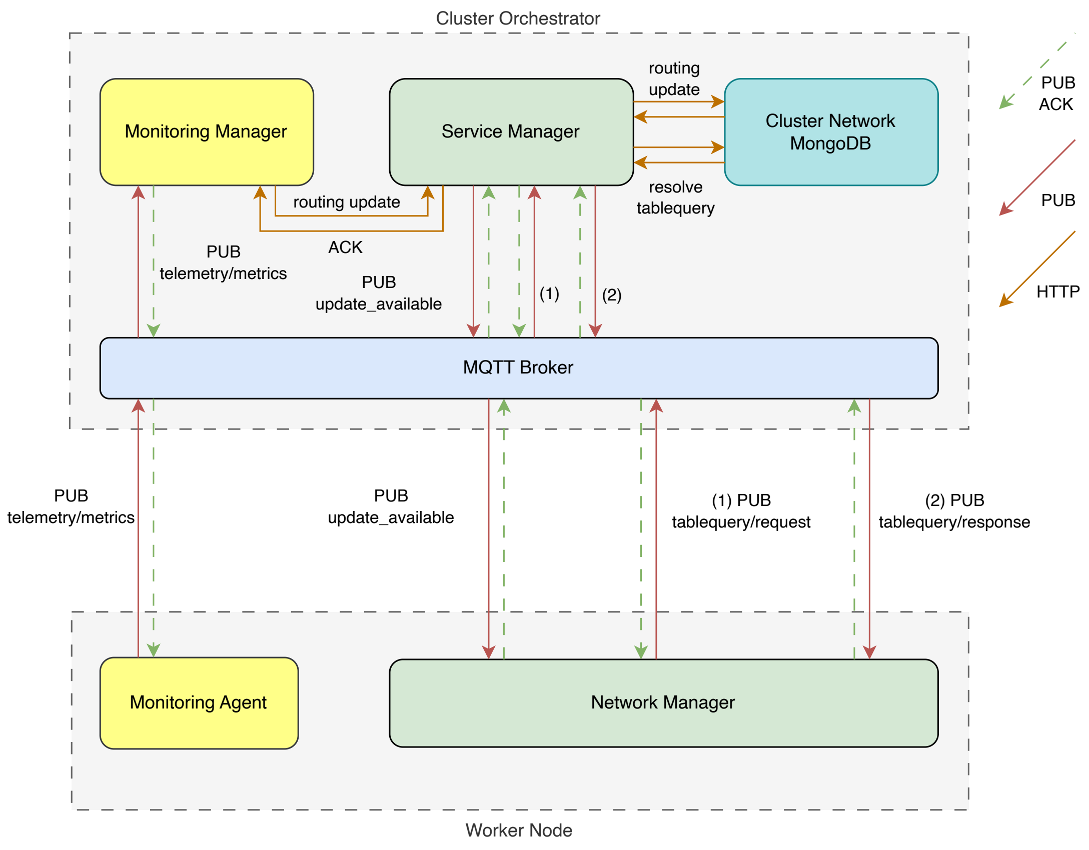

Figure 5.13.: Test 2: Application Scaling - Worker-Cluster Traffic Flow

in the baseline measurement. While running the test case, we observed a noticeable slowdown in the node engine's logging to the console with increasing application instances, which in combination with the very high CPU usage most likely is due to the fact that the log aggregation causes increased delays in the messaging process, which affect the measurement results. Nevertheless, we could not observe any delays or significant timing drifts for the remaining messages sent. This observation is also reflected in the received data at the cluster node, where we can almost completely cover the received data volume of 121.97KB/s with 30 application instances.

By our observations we can generally conclude that most transferred data volumes we derived from the MQTT broker traffic analysis cover a significant portion of the observed network traffic, meaning that there is a minimal amount of background traffic influencing the results. However, with the chosen test design, we cannot draw any meaningful conclusions about the performance of the monitoring platform, only point out certain limitations and identify bottlenecks, as they show a very unrealistic worst case scenario for the resource requirements of the monitoring platform. In order to assess the performance in a more realistic scenario, adjustments to the sending frequency, sending volumes of the telemetry data and the routing update frequency are required. Given that we currently send all telemetry data collected, which quickly accumulates to up to 105KB sent every second, there is a lot of room for improvement in this regard. One possible solution would be to only send the telemetry data for the application instances that are relevant for the fulfillment of the active monitoring policies, routing included. This can effectively be realized through programmable metric filter functions in the processing pipeline of the monitoring agent, which would greatly reduce the transferred data volume, while still allowing the monitoring platform to perform its routing analysis.

Another important aspect to consider is the amount of metadata being included in the telemetry data. By using the OpenTelemetry Protocol, we are bound to its messaging format and metadata fields, which are currently not used by the monitoring platform, ultimately resulting in unnecessary data overhead. While a custom message format with only the necessary fields would allow for an implementation to the monitoring platform's needs, it would require a significant amount of work, maintenance and testing at the cost of losing the vendor-agnostic, and widely used OpenTelemetry Protocol.

5.3.3. Test 3: Transition Latency

Our final test evaluates the transition latency of the routing update in the case, where a load-balancing change has to be propagated to the interested worker nodes. It allows us to assess the responsiveness of the monitoring platform to changes in the application

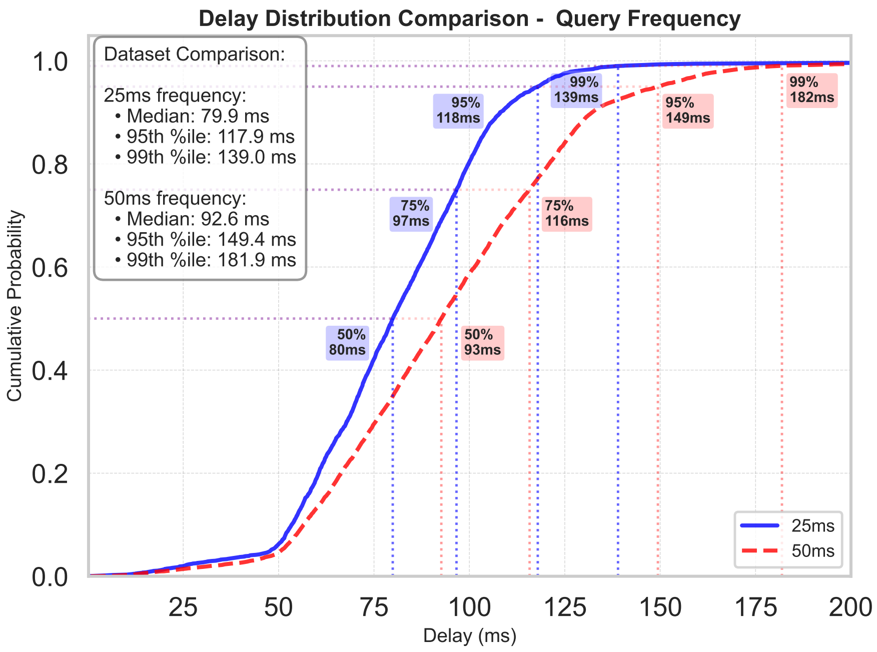

Figure 5.14.: Test 3: Transition Latency - Cumulative Distribution Function

state. Therefore, we provide our test results in the form of a Cumulative Distribution Function (CDF) in Figure 5.14 and additionally a boxplot in Figure 5.15, which visualize the distribution of the transition latency based on the frequency the client was querying the application.

For the test run with a 25ms query interval, we observed a median transition latency of 79.9ms with a 95th percentile of 117.9ms. With a 50ms query interval, we observed a slightly higher median transition latency of 92.5ms with a 95th percentile of 149.4ms. Based on the results it becomes apparent that the transition latency precision is dictated by the frequency of the client querying the application.

In our first runs of this test, we observed a very strict degradation in the observed transition latency of the application, which happened a few minutes into the test. We were able to trace this back to the flow identification mechanism in the network manager. It is responsible for identifying previously seen flows based on the 5-tuple of the TCP connection – source IP, destination IP, source port, destination port and protocol – in order to conduct the proxying of the client connection to the overlay network. In the first test runs, these flows were not being cleaned up, resulting in a growing number of collisions with old flow entries, which occurred based on the random source port selected by the client, as the remaining headers from the 5-tuple

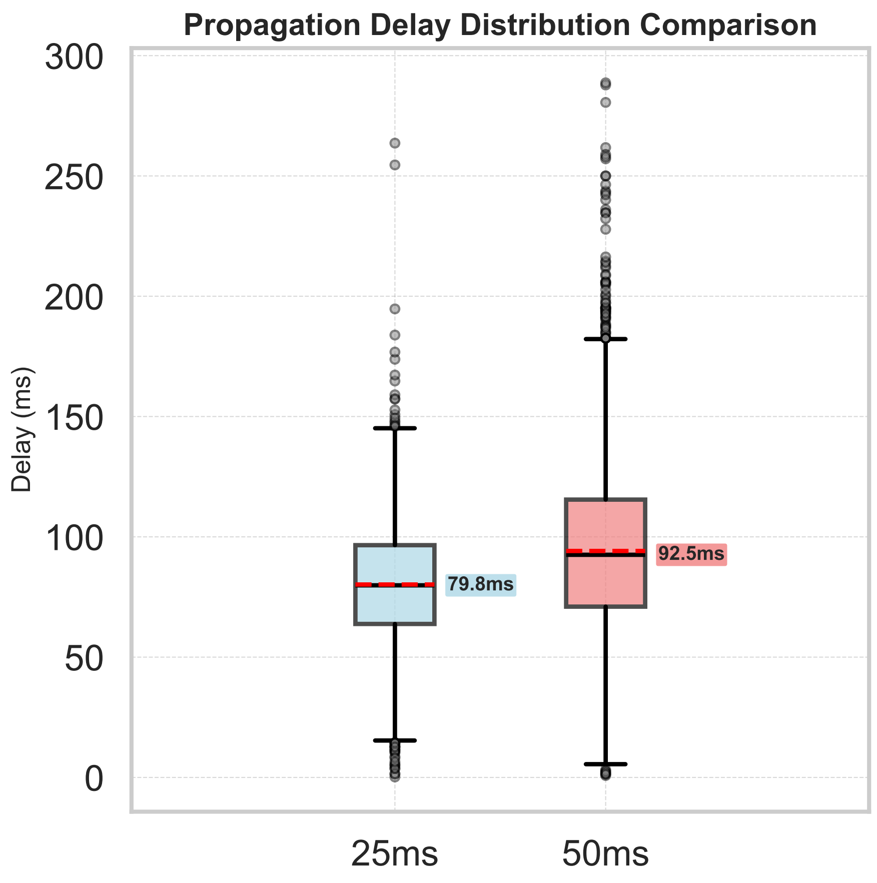

Figure 5.15.: Test 3: Transition Latency
Boxplot of Different Query Interval Tests

are static. Ultimately, this resulted in one or multiple subsequent requests being sent to the previously selected application instance in the erroneous old flow entry. To this end, we implemented a hotfix with a 1 minute flow timeout, which gets cleaned up by a background thread every 30 seconds. While this fix allowed us to conduct the test over prolonged periods of time, it still occurred from time to time, that a newly generated request was matching old flow entries, resulting in a higher transition latency based on our dataset analysis.

Based on the provided traffic flow visualization in Figure 5.13, we can see what operations are being performed in the backend, before the actual routing update arrives at the worker node in order to take effect. From the visualization, everything, with exception of the telemetry data propagation, is part of the routing update process. Excluding the initial alarm trigger, which is depicted as *routing update* in the Figure, we can see that there are three distinct MQTT signals being sent back and forth between the cluster orchestrator and the worker node. The MQTT message `tablequery/result` is the final message sent to the worker node, before the routing switch is applied by the network manager. In combination with the fact, that the tablequery in itself requires two separate calls to the MongoDB database, which we already identified as a possible performance bottleneck, it is highly likely that the timings can be improved by optimizing the database access.

On top of the delay caused by the message propagation, we need to also consider the delay caused by the proxying mechanism in the network manager as a possible reason for the observed transition latency. As every packet of a request has to pass through the proxy, which currently is implemented in user-space, it is highly likely that this can cause a noticeable delay in the transition latency analysis, due to frequent context switches in the proxy workflow.

Unfortunately, we were not able to commit to a more thorough timing analysis on the proposed reasons for the observed transition delay due to the time constraints, leaving this as a potential area for future work. Nonetheless, we find the transition latency to be satisfactory, w.r.t the requirements it was designed for.

5.3.4. Non-Functional Requirements Evaluation

In this section, we revisit the non-functional requirements we defined in Section 3.1 and evaluate the fulfillment of these requirements in our proposed solution based on the test results. We will focus on the scalability, performance, stability and lightweight requirements of the singular components of the monitoring platform.

NFR6 - Scalability

The scalability of the monitoring agent is a key requirement for the monitoring platform, as it is the main component running on the resource constrained worker nodes. In our testing we were able to observe that one of the most significant factors with the current implementation is the amount of memory and network usage is required for its operation. The results have shown that no performance degradation was observed in any of the different test cases, which is a good sign for the scalability of the monitoring agent. However the resource overhead will ultimately have to be revisited in the future, with a more optimized implementation of the monitoring agent.

The monitoring manager and routing manager have shown to scale very well throughout the testing, with no performance degradation or significant increase in resource usage, therefore fulfilling the scalability requirement.

NFR7 - Performance

Throughout our testing, we were not able to identify any slowdown of the system as a whole. All operations of our introduced monitoring components on worker and cluster were able to complete within the expected timeframes. No noticeable time drifts were observed, with the exception of the log aggregation in the node engine, indicating good performance of the introduced monitoring system.

NFR8 - Stability

In our testing we were not able to observe any network instabilities caused by the introduced telemetry in any of the introduced system components. Due to our chosen system design, we can with high degree of confidence guarantee the stability of the monitoring platform against network outages due to the choice of the asynchronous MQTT protocol, built for exactly this use case. This, however, remains to be tested in a real-world scenario, as the infrastructure we conducted the testing on is not representative for an unstable edge network environment.

NFR9 - Lightweight

As already pointed out in Section 5.3.1, the monitoring agent has shown to be quite demanding in terms of memory usage, caused by the `prometheusreceiver` module. As it is one of the key modules in the monitoring agent, a more lightweight implementation of the `prometheusreceiver` is advised as a replacement in order to reduce the memory footprint of the monitoring agent. With the current implementation, the monitoring agent was able to run reasonably lightweight on the worker nodes, with a maximum observed CPU usage of 7.22% with 30 application instances. With reasonable optimizations, and especially filtering of the scraped metrics, we imagine the agent to be able to drive the resource footprint to a very low level.

The monitoring and routing managers have shown to be very lightweight, considering the aggressive testing we conducted. As the network pressure is almost exclusively attributed to the telemetry data sent by the worker nodes, the resource footprint of the cluster-level monitoring components directly depends on the data volume, and thus the optimizations implemented in the monitoring agent.

6. Conclusion

6.1. Summary

The distributed monitoring platform presented in this thesis successfully implements the collection of cross-layer telemetry data about microservices deployed on orchestrated edge-cloud infrastructure, ultimately enabling telemetry-aware load balancing to monitored microservice replicas. The presented components are modular and resource-efficient, leveraging the state-of-the-art technologies and standards introduced by the OpenTelemetry project. By following design considerations of edge environments, the components utilize an event-driven architecture and asynchronous communication, offering resilient, reliable and performant messaging capabilities. The highly modular system design allows for easy integration of a wide range of industry respected monitoring systems, as well as the extension of the monitoring backend with new, user-defined custom metrics at runtime. Our monitoring backend establishes the groundwork for the implementation of an extensible framework for the programmatic definition of cluster-wide, telemetry-aware policies, further enhanced through a dynamic notification system. We demonstrate the validity of our overarching system design through a prototype policy implementation, which conducts load-aware service instance prioritization through mathematical modeling of load-balancing semantics on telemetry data. By means of worst-case scenario testing, scalability and performance concerns of the system were identified, analyzing the benefits and shortcomings of the proposed solution. Alongside remarkable route switching latency, the system design offers exceptional customizability and extensibility, providing a solid foundation for further research and development.

6.2. Limitations and Future Work

Thanks to the different testing scenarios conducted in the prototype evaluation, we were able to identify some limitations of the current system implementation, which need to be considered. Among these limitations, the most notable one is the quantity of telemetry data collected, resulting in a significant increase in network traffic. While the chosen messaging protocol itself is robust in edge environments, frequent packet loss would result in a significant degradation of the system's performance. Therefore, further research into the viability of the OpenTelemetry protocol in edge environments should be conducted. A more immediate solution would be the reduction of sent telemetry data, by means of intelligent filtering of the telemetry data, based on the application's and monitoring platform's requirements. One branch of research in the direction of intelligent filtering of telemetry data is the field of Age of Information (AoI) [61], which includes a measure of the freshness of the information received. The research indicates that sampling of telemetry data is not uniformly distributed, with certain metrics being more volatile and important than others, thus requiring more frequent sampling for better performance. Additionally, correlation techniques between metrics, as were already successfully applied in [60], could be used to reduce the amount of telemetry data sent. We are convinced that those techniques can find effective use in the context of the proposed monitoring platform and our identified limitations. Nonetheless, it remains apparent that telemetry inherently comes at the cost of system and network resources, which needs to be carefully considered in the operation of the system.

Another important aspect to the system design, which we addressed in the prototype implementation through an in-memory database, is the efficient storage and retrieval of telemetry data. While the performance benefits and quick implementation were the main driving factor for the chosen solution, we are convinced that the use of a time-series database would be beneficial for the long-term storage of telemetry data. Therefore, it needs to be evaluated what the performance and latency implications in the policy evaluation process are, as data would have to be retrieved over the network.

Finally, the proper implementation of the load-balancing algorithms is a topic that needs to be addressed in the future in order to move the system to a production-ready state. Our prototype implementation focused on the basic implementation of the aggregation and analysis of the telemetry data as a proof of concept. These basic features provide a solid foundation for the implementation of more complex load-balancing algorithms, evaluating system performance from a holistic perspective. What still remains to be implemented in the propagation of the load-balancing decisions from the cluster to the root orchestrator, which enables cross-cluster load-balanced traffic routing. An important aspect to consider to this end is the factor of data sovereignty, as the cluster needs to avoid the propagation of sensitive data across cluster boundaries. While our routing data design may potentially already adhere to this requirement, assuming we only propagate the service instance routing priority, it remains to be evaluated for the final implementation whether inference on sensitive data is possible.

A. Appendix

A.1. Monitoring Agent Configuration

A.1.1. Builder Manifest

This is the builder manifest as it is used to build the monitoring agent. It is used to define the modules to be included in the compiled binary. Extending the manifest is possible by adding the `gomod` directive to the `receivers`, `processors` and `exporters` sections. During build time, the manifest is then used to download the modules from the specified source and compile them into the binary. Therefore, they have to adhere to the OpenTelemetry Collector Component structure [9].

```yaml
  dist:
  #  module: github.com/smnzlnsk/monitoring-agent
  description: Master Thesis - Telemetry driven Network Optimization in
   ↪  Edge-Cloud Orchestration Frameworks
  otelcol_version: 0.109.0
  version: 0.1.0
  output_path: /tmp/monitoring-agent
  name: monitoringagent

receivers:
  - gomod:
   ↪  github.com/open-telemetry/opentelemetry-collector-contrib/receiver/
   ↪  hostmetricsreceiver v0.109.0
  - gomod:
   ↪  github.com/open-telemetry/opentelemetry-collector-contrib/receiver/
   ↪  prometheusreceiver v0.109.0
  - gomod: go.opentelemetry.io/collector/receiver/otlpreceiver v0.109.0

processors:
  - gomod:
   ↪  github.com/open-telemetry/opentelemetry-collector-contrib/processor/
   ↪  groupbyattrsprocessor v0.109.0
  - gomod:
  ↪  github.com/open-telemetry/opentelemetry-collector-contrib/processor/
  ↪  resourceprocessor v0.109.0
  - gomod:
  ↪  github.com/open-telemetry/opentelemetry-collector-contrib/processor/
  ↪  metricstransformprocessor v0.109.0
  - gomod: go.opentelemetry.io/collector/processor/batchprocessor
  ↪  v0.109.0

exporters:
  - gomod: github.com/smnzlnsk/opentelemetry-components/exporter/
  ↪  mqttexporter v0.1.0
```

A.1.2. OpenTelemetry Collector Configuration File

This is the configuration file for the OpenTelemetry Collector, which is passed to the monitoring agent at startup. It is used to configure the receivers, processors and exporters for the OpenTelemetry Collector.

```
receivers:
otlp:
  protocols:
    grpc:
      endpoint: 0.0.0.0:4317
    http:
      endpoint: 0.0.0.0:4318

prometheus:
  config:
    scrape_configs:
      - job_name: 'containerd-prometheus'
        scrape_interval: 1s
        static_configs:
          - targets: ['localhost:9323']
        metrics_path: /v1/metrics
        metric_relabel_configs:
          - source_labels: [namespace]
            regex: 'oakestra'
            action: keep
  hostmetrics:
    collection_interval: 1s
    #root_path: /
    scrapers:
      cpu:
        metrics:
          system.cpu.utilization:
            enabled: true
          system.cpu.logical.count:
            enabled: true
          system.cpu.physical.count:
            enabled: true
      memory:
        metrics:
          system.memory.limit:
            enabled: true

processors:
  metricstransform:
    transforms:
      - include: ^container_cpu_(?P<state>.*)_usec_microseconds$
        match_type: regexp
        action: combine
        new_name: container.cpu.time
        submatch_case: lower
      - include: ^container_memory_(?P<state>.*)_bytes
        match_type: regexp
        action: combine
        new_name: container.memory.usage
        submatch_case: lower
      - include: ^container_memory_oom_total
        match_type: regexp
        action: update
        new_name: container.memory.oom
        submatch_case: lower
      - include: (.*).cpu.*
        match_type: regexp
        action: update
        operations:
          - action: aggregate_labels
            label_set: [ state ]
            aggregation_type: sum
      - include: (.*).memory.*
        match_type: regexp
        action: update
        operations:
          - action: aggregate_labels
            label_set: [ state ]
            aggregation_type: sum
          - action: update_label
            label: state
            value_actions:
              - value: usage
                new_value: used

  groupbyattrs:
    keys:
      - container_id
      - namespace

  batch:
    timeout: 1s

  resource:
    attributes:
      - key: machine
        value: ${env:OTEL_RESOURCE_MACHINE_ID}
        action: upsert

exporters:
  mqtt:
    interval: 1s
    client_id: ${env:OTEL_RESOURCE_MACHINE_ID}
    broker:
      host: mqtt.broker.ip
      port: 10003

service:
```

```
pipelines:
  metrics:
    receivers: [prometheus, hostmetrics, otlp]
    processors: [resource, groupbyattrs, metricstransform, batch]
    exporters: [mqtt]
```

A.2. Enabling the containerd Prometheus Metrics Endpoint

This section provides a guide on how to enable the containerd Prometheus metrics endpoint for container metric reporting. This is required for the monitoring agent to be able to collect the metrics about the containerized applications.

```
# /etc/containerd/config.toml
[metrics]
address = "0.0.0.0:9323"
grpc_histogram = true

# bash command to restart containerd
sudo systemctl restart containerd
```

A.3. Example OpenTelemetry Metadata File

This section provides an example of the `metadata.yaml` file for the `cpu` subprocessor. This file is used to define the metrics and attributes that are collected by the subprocessor. Using `mdatagen`, the necessary code scaffolding is generated, which can be used to create and populate the metrics and attributes according to the defined schema.

```
# metadata.yaml
type: oakestraprocessor/cpu

parent: oakestraprocessor

resource_attributes:
  service.name:
    description: service name
    type: string
    enabled: true

  container_id:
    description: container id, which is the full service instance
     ↪  identifier
    type: string
    enabled: true

attributes:
  state:
    description: recorded states
    type: string
    enum: [system, user]

metrics:
  service.cpu.utilisation:
    enabled: true
    description: percentage of total cpu time used by service in
     ↪  certain state
    unit: percent
    gauge:
      value_type: double
    attributes: [state]
```

A.4. Example Formula

An example for a formula is given below. The formula is used to calculate the memory utilization of a service instance. The formula is used in the `memory` subprocessor, which is used to calculate the memory utilization of a service instance.

```
([container.memory.usage] / [system.memory.limit{default}]) * 100
```

A.5. Internal Metric Referencing Guide

This section provides a guide on how the monitoring manager references metrics internally. This guide can be used to write custom formulas for the monitoring manager or in case of a future implementation of a new subprocessor, which is not covered by the autogenerated code.

We reference metrics by three distinct segments, given in A.1.

Segment	Example	Default Value	Description
`Metric Name`	`host.memory.usage`	None	The name of the metric to reference.
`Metric Attribute`	`free`	`default`	The metric `state` attribute.
`Metric Age`	`2`	`0`	The age of the metric datapoint to reference.

Table A.1.: Monitoring Manager - Internal Metric Referencing Guide

Whenever a metric is used as an argument inside the mathematical formula, the monitoring manager always saves the full identifier of the metric. This is done for easier retrieval of the metric datapoint from the database, as they are stored using the full identifier with a rotating pointer to the newest datapoint (age = 0). In order to get the previous datapoint, the age is **incremented** by one, as the newest datapoint is always the one with age = 0, the one with age = 1 is the second newest, and so on. Additionally, as some metrics do have a `state` attribute, like the `container.memory.usage` metric, the attribute can be used to filter for the correct datapoint with matching state attribute. Otherwise, do not specify the attribute, as it will be ignored and default to the default attribute value.

The order of the segments is important and expected to follow the following format:

```
<metric_name>(<metric_age>){<metric_attribute>}
```

Going by the example in A.4, the first formula argument is translated to the following full metric identifier string in the database:

```
container.memory.usage(0){default}
```

Applying this to the full formula, this results in the following formula being stored internally in the monitoring manager:

```
([container.memory.usage(0){default}] /
 ↪  [system.memory.limit(0){default}]) * 100
```

Based on this guide, the following example formula could be used to calculate the delta between the current and previous datapoint of a metric, which can be used for the calculation of the rate of change of a metric:

```
[metric.example.name(0)] - [metric.example.name(1)]
```

A.6. Monitoring Agent - Collected Metrics

This section provides a list of metrics that are collected by the monitoring agent. It is important to note the metrics can contain multiple datapoints, which are different in their `state` attribute. Therefore, we advise to read the documentation of the respective receivers available at the OpenTelemetry Registry [16].

Receiver	Metric	Description
`prometheusreceiver`	`container.memory.usage`	Memory usage of the container in Bytes
	`container.memory.oom`	Number of times the container was OOM killed
	`container.cpu.time`	Total time in microseconds the container was using the CPU
`hostmetricsreceiver`	`system.cpu.time`	CPU time used by the host system in Jiffies
	`system.cpu.logical.count`	Number of logical CPUs on the host system
	`system.cpu.physical.count`	Number of physical CPUs on the host system
	`system.cpu.utilization`	Percentage of total CPU time used by the host system in the system and user state over the last monitoring period
	`system.memory.usage`	Memory usage of the host system in Bytes
	`system.memory.limit`	Memory limit of the host system in Bytes
	`system.memory.utilization`	Percentage of total memory used by the host system

Table A.2.: Monitoring Agent - Collected Metrics

B. Implementing a new Load-Balancing Algorithm

This chapter provides a thorough, step-by-step description on how we implemented the new load-balancing algorithms, which components are affected and how the new routing capabilities are enabled through the monitoring platform. It can be used as a reference for the developer to understand the new routing mechanisms, how to enable them and how to extend the system with new custom load-balancing algorithms. As the implementation details are already given in Chapter 4, we will keep this chapter short and concise, only focusing on the changes needed to be made to the backend components, in order to start using the new load-balancing algorithm.

B.1. System Manager Changes

B.1.1. Change the SLA Schema

Adjust the SLA schema to include fields for the user-defined reservation of new Service IPs.

```
# sla/schema.py
... # microservice context
"addresses": {
  "type": "object",
  "properties": {
      "rr_ip": {"type": "string"},
      "rr_ip_v6": {"type": "string"},
      "closest_ip": {"type": "string"},
      "closest_ip_v6": {"type": "string"},
      # TODO: add new fields here:
      "new_balancing_policy_ip": {"type": "string"},
      "new_balancing_policy_ip_v6": {"type": "string"},
      ############
      "instances": { ... },
  },
},
```

B.1.2. Change the service management handler

Add the new fields to the service management handler. This way, the information is available for the service manager to use.

```
def generate_db_structure(application, microservice):
  microservice["applicationID"] = application["applicationID"]
  microservice["app_name"] = application["application_name"]
  microservice["app_ns"] = application["application_namespace"]
  microservice["service_name"] = microservice["microservice_name"]
  microservice["service_ns"] = microservice["microservice_namespace"]
  microservice["image"] = microservice["code"]
  microservice["next_instance_progressive_number"] = 0
  microservice["instance_list"] = []
  if microservice["virtualization"] == "container":
      microservice["virtualization"] = "docker"
  addresses = microservice.get("addresses")
  if addresses is not None:
      microservice["RR_ip"] = addresses.get(
          "rr_ip"
      )  # compatibility with older netmanager versions
      microservice["RR_ip_v6"] = addresses.get("rr_ip_v6")
      # TODO: add here
      microservice["new_balancing_ip"] =
       ↪  addresses.get("new_balancing_policy_ip")
      microservice["new_balancing_ip_v6"] =
       ↪  addresses.get("new_balancing_policy_ip_v6")
      #######
  if microservice["virtualization"] == "unikernel":
      microservice["arch"] = microservice["arch"]
  return microservice
```

B.2. Root Service Manager Changes

B.2.1. Implement Service IP Management

Implement the validation logic for the new balancing IPv6 addresses. The address space reservation needs to be approved by the maintainers.

```
# interfaces/mongodb/validators.py
  def validate_new_balancing_policy_ipv6(address):
    validate_ipv6_length(address) # validate the length of the address
    assert address[0] == 253
    assert address[1] == 255
    assert address[2] == 128 # <-- this is the IPv6 subnet identifier for
    ↪  the new balancing policy
    assert 0 <= address[3] <= 8
```

B.2.2. Implement database wrapper functions

In order to allow for the database IP address management of the new balancing policy, we need to implement a set of wrapper functions. Create a new file `<new-policy>.py` in the `interfaces/mongodb` directory, and implement functions following the naming convention:

- `mongo_get_<new-policy-name>_address_from_cache_v6` - Get the next available IP address from the cache
- `mongo_free_<new-policy-name>_address_to_cache_v6` - Free the IP address and add it to the cache
- `mongo_get_next_<new-policy-name>_ip_v6` - Get the next available IP address from the database
- `mongo_update_next_<new-policy-name>_ip_v6` - Update the next available IP address in the database

For easier development, take motivation from the existing balancing policy implementations, such as `rr.py` and `closest.py`.

B.2.3. Integrate database wrapper functions in the abstracted database requests

With the wrapper functions implemented, we need to integrate those into the abstracted database request function. This is done by adding the new policy name to the default address map in the `_mongo_ip_operation` function, referenced by a new constant.

```
# interfaces/mongodb/requests.py
  # 1. At the very top of the file, create a new constant for your new
  ↪  policy
```

```
... # existing constants
ADDR_NEW_BALANCING = "<new-policy-name>"

... # existing code
# 2. Add the first address from your reserved IPv6 address space to the
↪  default_addr_map in the _mongo_ip_operation function:
... # existing code
default_addr_map = {
  ...
  IP_V6: {
    ADDR_NEW_BALANCING: [253, 255, 128, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    ↪  0, 0] # <-- this is the first address from your reserved IPv6
    ↪  address space
  }
}
...
```

B.2.4. Implement the network management logic

In order to make the new balancing policy available to be managed by the service manager, we need to first implement the new policy network management logic for IPv6 addresses. The following code snippet shows the function signatures needed to be implemented as an override to the `IPv6InstanceStrategy` class.

```
# network/management/service/v6/new_balancing_policy.py
class IPv6NewBalancingPolicyStrategy(IPv6InstanceStrategy):
    """IPv6 new balancing policy address allocation strategy"""

    def validate_custom_address(self, address: str, job_name: str) ->
    ↪  bool:
    def get_next_address(self) -> str:
    def clear_address(self, address: str) -> None:
    def _increase_address(self, addr: List[int]) -> List[int]:
```

Tip: Take inspiration from the existing balancing policy implementations and adjust to your network management logic. The previously implemented database wrapper functions are also encouraged to be used.

B.2.5. Map the new strategy to the balancing policy

In order for the address manager to be able to assign new addresses to the service instances based on the new addressing strategy, we need to create the mapping. Therefore, extend the `self.strategies` dictionary in the `IPAddressManager` `__init__` function.

```
# network/management/service/v6/manager.py
... # existing code
self.strategies = {
  ... # existing strategies
  STRATEGY_IPV6_NEW_BALANCING: IPv6NewBalancingPolicyStrategy(), # <--
   ↪  this is the new strategy
}
... # existing code
```

B.2.6. Assign new addresses to the service instances

Now that the `IPAddressManager` is aware of the new strategy, we need to hand out new addresses to the service instances. This is achieved by extending the `s_ip` array in the `deploy_request()` deployment handler.

```
# operations/service_management.py
... # existing code
s_ip = [{ ... }, # existing address assignments
  {
    "IpType": 'NewBalancingPolicy',
    "Address": ip_manager.new_address(
      STRATEGY_IPV4_SERVICE,
      deployment_descriptor.get('new_balancing_ip'), # <- this is the
       ↪  field we added to the service schema
      job_name
    ),
    "Address_v6": ip_manager.new_address(
      STRATEGY_IPV6_NEW_BALANCING, # <-- this is the new strategy
       ↪  reference we added to the IPAddressManager
      deployment_descriptor.get('new_balancing_ip_v6'), # <- this is the
       ↪  field we added to the service schema
      job_name),
  }
]
... # existing code
```

This can also be done conditionally, if you wish to implement a routing policy that is limited to a specific functional requirement of the application.

```
# operations/service_management.py
... # existing code
s_ip = [{ ... }] # default address assignments
monitoring_class = deployment_descriptor.get('monitoring_class')
# you conditional address assignments here
if monitoring_class is not None and monitoring_class ==
↪  'your_new_monitoring_class':
  s_ip.append({
    "IpType": 'NewBalancingPolicy',
    "Address": ip_manager.new_address(STRATEGY_IPV4_SERVICE,
     ↪  deployment_descriptor.get('new_balancing_ip'), job_name),
    "Address_v6": ip_manager.new_address(
      STRATEGY_IPV6_NEW_BALANCING,
      deployment_descriptor.get('new_balancing_ip_v6'),
      job_name
    ),
  })
... # existing code
```

B.2.7. What about IPv4?

The IPv4 address space is shared by all balancing policies, so they do not require any special changes. This is only the case for IPv6, where the address spaces are dedicated to each balancing policy and a design choice by the Oakestra platform [26].

B.3. Monitoring Manager Changes

With the service manager now handing out new Service IPs, with the goal of implementing a new balancing policy, we need to implement the new balancing policy logic in the monitoring manager. Because the routing manager will become aware on deployment that the new Service IP type is present for this service, it will start query the monitoring manager's routing policy evaluator under that specific new Service IP type. In this particular example we have called the Service IP type `NewBalancingPolicy`, therefore we will have to implement a new evaluator under the same identifier, in order to query the new balancing policy logic.

B.3.1. Implement the new evaluator

Inside the *oakestraheuristicengine* module, we create the new evaluator, implementing the `Evaluator` interface. For the routing policy evaluator, we place them under the evaluator collection directory, which contains a collection of existing evaluators and can be used as a structural reference.

B.3.2. Add the new evaluator to the collection in the routing policy

Edit the `routing.go` file, more specifically the `NewRoutingEntity` function, and add the new evaluator to the collection.

```go
// internal/heuristicentity/entities/routing/routing.go

func NewRoutingEntity(services domain.Services, logger *zap.Logger)
 ↪  domain.HeuristicEntity {
        processorStore := processor.NewStore()

        // TODO: Add processors here

        // First initialize the processors evaluator
  /* existing evaluators */

  newBalancingEvaluator :=
   ↪  evaluators.NewNewBalancingPolicyEvaluator("NewBalancingPolicy")

        // Create and add the processors to the processor store
  /* existing processors */
        processorStore.Add(processor.NewProcessor("NewBalancingPolicy",
         ↪  newBalancingEvaluator))

        return &routingEntity{
                ...
        }
}
```

Notice how we passed the "NewBalancingPolicy" string as the processor name. This is the identifier that the routing manager will use to query the new evaluator. it is therefore important, to match those identifiers with the Service IP type name.

B.4. Network Manager Changes

With a new Service IP type being present in the system, we need to integrate the Network Manager, such that he is able to interpret the new Service IP type and route the traffic accordingly.

B.4.1. Add the new Service IP type to the Table Entry Cache

```go
// TableEntryCache/service.go

type ServiceIpType int

const (
  ... // other service ip types
  NewBalancingPolicy ServiceIpType = iota + 1
)

// implement Stringer interface
func (s ServiceIpType) String() string {
  return []string{
    ... // other service ip types
    "NewBalancingPolicy",
  }[s]
}

func toServiceIP(Type string, Addr string, Addr_v6 string) ServiceIP {
  ip := ServiceIP{
    IpType: 0,
    Address: Addr,
    Address_v6: Addr_v6,
  }

  switch Type {
    case "NewBalancingPolicy":
      ip.IpType = NewBalancingPolicy
    case ... // other service ip types
    default:
      ip.IpType = 0
  }
  return ip
}

func ServiceIpTypeFromString(s string) ServiceIpType {
  switch s {
    case "NewBalancingPolicy":
      return NewBalancingPolicy
    case ... // other service ip types2
    default:
      return 0
  }
}
```

With all those changes, the new balancing policy is now available to be used in the system.

C. Example Instrumented Application

This chapter provides a small example of an instrumented application, which can be used as a reference for a developer to understand how to instrument an application for the monitoring platform, especially in the Oakestra orchestration platform. The application is a simple Golang application, using the OpenTelemetry Golang SDK to instrument the application. This is of course not any production ready application code, but a small example to get the developer started with the OpenTelemetry SDK.

```go
package main

import (
        "context"
        "fmt"
        "log"
        "os"
        "os/signal"
        "time"

        "go.opentelemetry.io/otel"
        "go.opentelemetry.io/otel/attribute"
        "go.opentelemetry.io/otel/exporters/otlp/otlpmetric/otlpmetricgrpc"
        "go.opentelemetry.io/otel/metric"
        sdkmetric "go.opentelemetry.io/otel/sdk/metric"
        "go.opentelemetry.io/otel/sdk/resource"
        semconv "go.opentelemetry.io/otel/semconv/v1.24.0"
)

var (
        serviceName    = "basic-otel-app"
        serviceVersion = "1.0.0"
        containerID    = os.Getenv("HOSTNAME") // IMPORTANT: This is
         ↪  autoconfigured by the node engine, do not change this
```

```
  otlpEndpoint   = os.Getenv("OTEL_EXPORTER_OTLP_ENDPOINT") // IMPORTANT:
   ↪  This is autoconfigured by the SDK based on the environment
   ↪  variables
)

// Global metric instruments
var (
        exampleGauge metric.Float64ObservableGauge
)

func initMetrics(ctx context.Context) (func(context.Context) error, error)
 ↪  {
        // Create resource with service information
        res, err := resource.New(ctx,
                resource.WithAttributes(
                        semconv.ServiceName(serviceName),
                        semconv.ServiceVersion(serviceVersion),
                        semconv.HostName(containerID),
      attribute.String("container_id", containerID), // IMPORTANT: This
       ↪  is used to identify the container in the monitoring platform
                ),
        )
        if err != nil {
                return nil, fmt.Errorf("failed to create resource: %w",
                 ↪  err)
        }

        // Create OTLP gRPC exporter
        exporter, err := otlpmetricgrpc.New(ctx,
                otlpmetricgrpc.WithEndpoint(otlpEndpoint), // IMPORTANT:
                 ↪  You must either set this explicitly like we do here,
                 ↪  or not call it at all (because of autoconfiguration)
                otlpmetricgrpc.WithInsecure(),
        )
        if err != nil {
                return nil, fmt.Errorf("failed to create OTLP exporter:
                 ↪  %w", err)
        }
```

```
// Create metric provider
provider := sdkmetric.NewMeterProvider(
        sdkmetric.WithResource(res),
        sdkmetric.WithReader(sdkmetric.NewPeriodicReader(exporter,
                sdkmetric.WithInterval(5*time.Second),
        )),
)

// Set global meter provider
otel.SetMeterProvider(provider)

// Create meter
meter := provider.Meter(
        "basic-monitoring",
        metric.WithInstrumentationVersion(serviceVersion),
)

// Create example observable gauge
exampleGauge, err = meter.Float64ObservableGauge(
        "your_metric_name",
        metric.WithDescription("your_metric_description"),
        metric.WithUnit("%"),
)
if err != nil {
        return nil, fmt.Errorf("failed to create example gauge:
         ↪  %w", err)
}

// Register callback for observable gauges
_, err = meter.RegisterCallback(
        func(ctx context.Context, observer metric.Observer) error
         ↪  {
                observer.ObserveFloat64(exampleGauge, 100.0,
                 ↪  metric.WithAttributes(
                        attribute.String("state",
                         ↪  "your_state_string_here"),
                ))
                return nil
        },
                exampleGauge,
        )
        if err != nil {
                return nil, fmt.Errorf("failed to register callback: %w",
                 ↪  err)
        }

        return func(ctx context.Context) error {
                return provider.Shutdown(ctx)
        }, nil
}

func collectMetrics() {
        // This is where you would normally collect real metrics from the
         ↪  system
        // or replace this with your own metric collection logic
}

func main() {
        // Create context with signal handling
        ctx, cancel := signal.NotifyContext(context.Background(),
         ↪  os.Interrupt)
        defer cancel()

        // Initialize metrics
        shutdown, err := initMetrics(ctx)
        if err != nil {
                log.Fatalf("Failed to initialize metrics: %v", err)
        }
        defer func() {
                if err := shutdown(context.Background()); err != nil {
                        log.Printf("Error during shutdown: %v", err)
                }
        }()

        // Start metric collection loop
        ticker := time.NewTicker(1 * time.Second)
        defer ticker.Stop()
        for {
                select {
                case <-ctx.Done():
                        log.Println("Shutting down...")
                        return
                case <-ticker.C:
                        collectMetrics()
                }
        }
}
```

It is important to point out, that the `hostname` and `otlpEndpoint` are being injected by the node engine into the container, so the developer does not need to explicitly set them. The `containerID` is used to identify the container in the monitoring platform, and is also being injected by the node engine, only required to be set if the developer wants to instrument the application with custom metrics.

D. Example Deployment Descriptors

D.1. Application with Load-Aware Routing Utilization

This section provides a small example of a deployment descriptor, which can be used as a reference for the developer to deploy an application and use the new load-balancing algorithms for his application routing.

```
{
  "sla_version" : "v2.0",
  "customerID" : "Admin",
  "applications" : [
    {
      "applicationID" : "",
      "application_name" : "clientserver",
      "application_namespace" : "test",
      "application_desc" : "Simple demo with curl client and NGINX
      ↪  server",
      "microservices" : [
        {
          "microserviceID": "",
          "microservice_name": "curl",
          "microservice_namespace": "test",
          "virtualization": "container",
          "cmd": ["sh", "-c", "curl 10.30.5.1"],
          "memory": 100,
          "vcpus": 1,
          "vgpus": 0,
          "vtpus": 0,
          "bandwidth_in": 0,
          "bandwidth_out": 0,
          "storage": 0,
          "code": "docker.io/curlimages/curl:7.82.0",
          "state": "",
          "port": "9080",
          "added_files": []
        },
        {
          "microserviceID": "",
          "microservice_name": "nginx",
          "microservice_namespace": "test",
          "virtualization": "container",
          "cmd": [],
          "memory": 100,
          "vcpus": 1,
          "vgpus": 0,
          "vtpus": 0,
          "bandwidth_in": 0,
          "bandwidth_out": 0,
          "storage": 0,
          "code": "docker.io/library/nginx:latest",
          "state": "",
          "port": "",
          "addresses": {
            "underutilized_ip": "10.30.5.1",
            "underutilized_ip_v6": "fdff:3000::501"
          },
          "added_files": []
        }
      ]
    }
  ]
}
```

By using the provided deployment descriptor, two microservices become available for deployment. A NGINX server, which based on the `underutilized_ip` directive reserves the Service IP `10.30.5.1` and Service IP `fdff:3000::501` for the `underutilized` load-balancing algorithm. Alongside, we provide the deployment definition for a curl client microservice, who starts a `HTTP GET` request to `10.30.5.1`, which is the Service IP reserved by the NGINX server [44], and shuts down after the request is completed. Assuming that we have multiple such NGINX server replicas deployed, the request against this specific Service IP will be resolved, proxied, and load-balanced by the network manager, based on the utilization evaluation of the NGINX server instances in the monitoring manager.

D.2. Application with Custom Metrics

This section provides a deployment descriptor of an application, which additionally requests the calculation of custom metrics for this application's microservice and its instances.

```
{
"sla_version" : "v2.0",
"customerID" : "Admin",
"applications" : [
  {
    "applicationID" : "",
    "application_name" : "clientserver",
    "application_namespace" : "test",
    "application_desc" : "Simple demo with curl client and NGINX
    ↪  server",
    "microservices" : [
      {
        "microserviceID": "",
        "microservice_name": "nginx",
        "microservice_namespace": "test",
        "virtualization": "container",
        "cmd": [],
        "memory": 100,
        "vcpus": 1,
        "vgpus": 0,
        "vtpus": 0,
        "bandwidth_in": 0,
        "bandwidth_out": 0,
        "storage": 0,
        "code": "docker.io/library/nginx:latest",
        "state": "",
        "port": "",
        "added_files": [],
        "monitoring": [
          {
            "output_metric_name": "service.memory.delta",
            "formula": "[container.memory.usage(1)] -
            ↪  [container.memory.usage(0)]",
            "states": ["used"],
              "description": "The delta of the memory usage of the
               ↪  service instance."
            }
          ]
        }
      ]
    }
  ]
}
```

The monitoring directive is used to define the developer requested metrics for calculation. It would be included in the notification message for the `oakestraprocessor` monitoring module in the monitoring manager, more specifically the `application` subprocessor, who would set up the workflow needed to calculate and enrich the telemetry data with the requested metrics.

List of Figures

2.1. General System Architecture of a Monitoring Platform 4
2.2. Example of a OpenTelemetry Collector Pipeline 8
2.3. Simplified Architecture of a Load-Balanced Service 9
2.4. Oakestra System Architecture 13

3.1. General Orchestration Platform Architecture 20
3.2. General Orchestration Platform Architecture with Telemetry-Driven Load-Balancing 23

4.1. General Oakestra System Architecture with Integrated Monitoring System 25
4.2. Monitoring Agent Details 27
4.3. Monitoring Manager Details 31
4.4. Contract Interface Workflow in Oakestra Processor 34
4.5. Routing Manager Details 37

5.1. Test 1: Node Scaling - CPU Usage 48
5.2. Test 1: Node Scaling - Memory Usage 49
5.3. Test 1: Node Scaling - Network Usage - Packets Received per Second . 51
5.4. Test 1: Node Scaling - Network Usage - Packets Sent per Second 51
5.5. Test 1: Node Scaling - Network Usage - KB/s Received 53
5.6. Test 1: Node Scaling - Network Usage - KB/s Sent 53
5.7. Test 2: Application Scaling - CPU Usage 55
5.8. Test 2: Application Scaling - Memory Usage 57
5.9. Test 2: Application Scaling - Network Usage - Packets Received per Second 58
5.10. Test 2: Application Scaling - Network Usage - Packets Sent per Second 58
5.11. Test 2: Application Scaling - Network Usage - KB/s Received 59
5.12. Test 2: Application Scaling - Network Usage - KB/s Sent 59
5.13. Test 2: Application Scaling - Worker-Cluster Traffic Flow 62
5.14. Test 3: Transition Latency - Cumulative Distribution Function 64
5.15. Test 3: Transition Latency Boxplot of Different Query Interval Tests . . . 65

List of Tables

3.1. Non-Functional Requirements Overview 19
3.2. Functional Requirements for the Monitoring Backend 20

4.1. Monitoring Agent - OpenTelemetry Collector Pipeline Modules 27
4.2. Comparative Analysis of Observability Backends 30
4.3. Monitoring Manager - OpenTelemetry Collector Pipeline Modules . . . 32
4.4. FPS Load Balancing Algorithm - Metrics Working Set 42

5.1. Fixed Test Parameters 45
5.2. Worker Node Network Traffic Patterns 52
5.3. Test 1: Node Scaling - Cluster Orchestrator Network Traffic 54
5.4. MQTT Broker Observed Traffic Patterns 61

A.1. Monitoring Manager - Internal Metric Referencing Guide 77
A.2. Monitoring Agent - Collected Metrics 79

Bibliography

[1] containerd Authors. *containerd. An industry-standard container runtime with an emphasis on simplicity, robustness and portability*. URL: `https://containerd.io`. (accessed: 06/20/2025).

[2] eBPF.io Authors. *eBPF. Dynamically program the kernel for efficient networking, observability, tracing and security*. URL: `https://ebpf.io`. (accessed: 07/05/2025).

[3] E. P. Authors. *Supported load balancers*. URL: `https://www.envoyproxy.io/docs/envoy/v1.34.1/intro/arch_overview/upstream/load_balancing/load_balancers#arch-overview-load-balancing-types`. (accessed: 06/22/2025).

[4] E. P. Authors. *What is Envoy*. URL: `https://www.envoyproxy.io/docs/envoy/v1.34.1/intro/what_is_envoy`. (accessed: 06/22/2025).

[5] G. Authors. *Build simple, secure, scalable system with Go*. URL: `https://go.dev`. (accessed: 06/21/2025).

[6] J. Authors. *Jaeger: open-source, distributed tracing platform*. URL: `https://www.jaegertracing.io`. (accessed: 06/25/2025).

[7] O. Authors. *A Lightweight Hierarchical Orchestration Framework for Edge Computing*. URL: `https://github.com/oakestra/oakestra`. (accessed: 07/06/2025).

[8] O. Authors. *Building a custom collector*. URL: `https://opentelemetry.io/docs/collector/custom-collector/`. (accessed: 06/27/2025).

[9] O. Authors. *Building custom components*. URL: `https://opentelemetry.io/docs/collector/building/`. (accessed: 07/12/2025).

[10] O. Authors. *Collector*. URL: `https://opentelemetry.io/docs/collector/`. (accessed: 07/15/2025).

[11] O. Authors. *Logs*. URL: `https://opentelemetry.io/docs/concepts/signals/logs/`. (accessed: 06/20/2025).

[12] O. Authors. *mdatagen*. URL: `https://github.com/open-telemetry/opentelemetry-collector/tree/main/cmd/mdatagen`. (accessed: 06/29/2025).

[13] O. Authors. *Metrics*. URL: `https://opentelemetry.io/docs/concepts/signals/metrics/`. (accessed: 06/20/2025).

[14] O. Authors. *OpenTelemetry*. URL: `https://opentelemetry.io`. (accessed: 06/21/2025).

[15] O. Authors. *OpenTelemetry SDKs*. URL: `https://opentelemetry.io/docs/languages/`. (accessed: 06/22/2025).

[16] O. Authors. *Registry. Find libraries, plugins, integrations, and other useful tools for using and extending OpenTelemetry.* URL: `https://opentelemetry.io/ecosystem/registry/`. (accessed: 06/21/2025).

[17] O. Authors. *Zero-code*. URL: `https://opentelemetry.io/docs/concepts/instrumentation/zero-code/`. (accessed: 06/22/2025).

[18] O. Authors. *OpenTracing*. URL: `https://opentracing.io`. (accessed: 06/21/2025).

[19] P. Authors. *Data model*. URL: `https://prometheus.io/docs/concepts/data_model/`. (accessed: 06/21/2025).

[20] P. Authors. *Node Exporter Repository*. URL: `https://github.com/prometheus/node_exporter`. (accessed: 06/28/2025).

[21] P. Authors. *OpenMetrics*. URL: `https://github.com/prometheus/OpenMetrics`. (accessed: 06/21/2025).

[22] P. Authors. *Overview. What is Prometheus?* URL: `https://prometheus.io/docs/introduction/overview/`. (accessed: 06/21/2025).

[23] P. Authors. *Prometheus Pushgateway Repository*. URL: `https://github.com/prometheus/pushgateway`. (accessed: 06/20/2025).

[24] T. J. Authors. *Agent*. URL: `https://www.jaegertracing.io/docs/1.24/architecture/#agent`. (accessed: 06/28/2025).

[25] T. K. Authors. *kube-proxy*. URL: `https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/`. (accessed: 06/25/2025).

[26] T. O. Authors. *IPv6 addressing*. URL: `https://www.oakestra.io/docs/manuals/networking-internals/ipv6-addressing/`. (accessed: 07/13/2025).

[27] J.-y. Baek, G. Kaddoum, S. Garg, K. Kaur, and V. Gravel. “Managing Fog Networks using Reinforcement Learning Based Load Balancing Algorithm.” In: *2019 IEEE Wireless Communications and Networking Conference (WCNC)*. 2019, pp. 1–7. DOI: `10.1109/WCNC.2019.8885745`.

[28] R. Brondolin and M. D. Santambrogio. “A Black-box Monitoring Approach to Measure Microservices Runtime Performance.” In: *ACM Trans. Archit. Code Optim.* 17.4 (Nov. 2020). ISSN: 1544-3566. DOI: `10.1145/3418899`.

[29] K. Cao, Y. Liu, G. Meng, and Q. Sun. “An Overview on Edge Computing Research.” In: *IEEE Access* 8 (2020), pp. 85714–85728. DOI: `10.1109/ACCESS.2020.2991734`.

[30] Y. Desmouceaux, P. Pfister, J. Tollet, M. Townsley, and T. Clausen. “6LB: Scalable and Application-Aware Load Balancing with Segment Routing.” In: *IEEE/ACM Transactions on Networking* 26.2 (2018), pp. 819–834. DOI: `10.1109/TNET.2018.2799242`.

[31] Elastic. *Elasticsearch System Requirements*. URL: `https://discuss.elastic.co/t/hardware-requirements-good-resilency-elastic-8-4/314090`. (accessed: 06/29/2025).

[32] C. N. C. Foundation. *Observability*. URL: `https://glossary.cncf.io/observability/`. (accessed: 06/20/2025).

[33] T. L. Foundation. *OpenAPI Specification v3.1.0*. URL: `https://spec.openapis.org/oas/v3.1.0.html`. (accessed: 07/06/2025).

[34] B. Haberman and B. Hinden. *Unique Local IPv6 Unicast Addresses*. RFC 4193. Oct. 2005. DOI: `10.17487/RFC4193`.

[35] HAProxy. *haproxy*. URL: `https://github.com/haproxy/haproxy/`. (accessed: 06/22/2025).

[36] HAProxy. *HAProxy Kubernetes Ingress Controller*. URL: `https://github.com/haproxytech/kubernetes-ingress`. (accessed: 06/25/2025).

[37] HAProxy. *How HAProxy works*. URL: `https://docs.haproxy.org/3.2/intro.html#3.2`. (accessed: 06/23/2025).

[38] HAProxy. *Using ACLs and fetching samples*. URL: `https://www.haproxy.com/documentation/haproxy-configuration-manual/latest/#7`. (accessed: 06/23/2025).

[39] D. Inc. *agent-payload*. URL: `https://github.com/DataDog/agent-payload`. (accessed: 06/21/2025).

[40] D. Inc. *Datadog Agent*. URL: `https://github.com/DataDog/datadog-agent`. (accessed: 06/21/2025).

[41] D. Inc. *Datadog. Modern monitoring & security. See inside any stack, any app, at any scale, anywhere.* URL: `https://www.datadoghq.com`. (accessed: 06/21/2025).

[42] D. Inc. *Develop faster. Run anywhere.* URL: `https://www.docker.com`. (accessed: 06/25/2025).

[43] D. Inc. *Install Docker Engine*. URL: `https://docs.docker.com/engine/install/`. (accessed: 07/06/2025).

[44] D. Inc. *Nginx Docker Image*. URL: `https://hub.docker.com/_/nginx`. (accessed: 07/06/2025).

[45] A. Khiyaita, H. E. Bakkali, M. Zbakh, and D. E. Kettani. "Load balancing cloud computing: State of art." In: *2012 National Days of Network Security and Systems*. 2012, pp. 106–109. DOI: 10.1109/JNS2.2012.6249253.

[46] S. Krishnan. *Segment Routing over IPv6 (SRv6) Segment Identifiers in the IPv6 Addressing Architecture*. RFC 9602. Oct. 2024. DOI: 10.17487/RFC9602.

[47] P. Kumar and R. Kumar. "Issues and Challenges of Load Balancing Techniques in Cloud Computing: A Survey." In: *ACM Comput. Surv.* 51.6 (Feb. 2019). ISSN: 0360-0300. DOI: 10.1145/3281010.

[48] J. Levin and T. A. Benson. "ViperProbe: Rethinking Microservice Observability with eBPF." In: *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. 2020, pp. 1–8. DOI: 10.1109/CloudNet51028.2020.9335808.

[49] G. LLC. *Protocol Buffers*. URL: https://protobuf.dev. (accessed: 07/04/2025).

[50] H. T. LLC. *HAProxy Enterprise*. URL: https://www.haproxy.com/documentation/haproxy-enterprise/. (accessed: 06/22/2025).

[51] S. K. Mishra, B. Sahoo, and P. P. Parida. "Load balancing in cloud computing: A big picture." In: *Journal of King Saud University - Computer and Information Sciences* 32.2 (2020), pp. 149–158. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2018.01.003.

[52] R. Moskowitz, D. Karrenberg, Y. Rekhter, E. Lear, and G. J. de Groot. *Address Allocation for Private Internets*. RFC 1918. Feb. 1996. DOI: 10.17487/RFC1918.

[53] E. Mosquitto. *Eclipse Mosquitto. An open source MQTT broker.* URL: https://mosquitto.org. (accessed: 06/27/2025).

[54] MQTT. *MQTT Specification*. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html. (accessed: 06/27/2025).

[55] M. Otero, J. M. Garcia, and P. Fernandez. "Towards a lightweight distributed telemetry for microservices." In: *2024 IEEE 44th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 2024, pp. 75–82. DOI: 10.1109/ICDCSW63686.2024.00018.

[56] M. Satyanarayanan. "The Emergence of Edge Computing." In: *Computer* 50.1 (2017), pp. 30–39. DOI: 10.1109/MC.2017.9.

[57] D. Stenberg. *curl*. URL: https://curl.se. (accessed: 07/06/2025).

[58] O. Team. *Elasticsearch Minimum Requirements*. URL: https://opster.com/guides/elasticsearch/capacity-planning/elasticsearch-minimum-requirements/. (accessed: 06/29/2025).

[59] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos. "Challenges and Opportunities in Edge Computing." In: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. 2016, pp. 20–26. DOI: `10.1109/SmartCloud.2016.18`.

[60] Z. Yao, Y. Desmouceaux, J.-A. Cordero-Fuertes, M. Townsley, and T. Clausen. "HLB: Toward Load-Aware Load Balancing." In: *IEEE/ACM Transactions on Networking* 30.6 (2022), pp. 2658–2673. DOI: `10.1109/TNET.2022.3177163`.

[61] R. D. Yates, Y. Sun, D. R. Brown, S. K. Kaul, E. Modiano, and S. Ulukus. "Age of Information: An Introduction and Survey." In: *IEEE Journal on Selected Areas in Communications* 39.5 (2021), pp. 1183–1210. DOI: `10.1109/JSAC.2021.3065072`.

[62] S. Zelenski. *Benchmarking Containers*. URL: `https://github.com/smnzlnsk/master-thesis-containers`. (accessed: 07/13/2025).

[63] S. Zelenski. *Master Thesis Artifacts - Oakestra Main Component*. URL: `https://github.com/smnzlnsk/mt-tum-artifacts-telemetry-network-optimization-main`. (accessed: 07/14/2025).

[64] S. Zelenski. *Master Thesis Artifacts - Oakestra Network Component*. URL: `https://github.com/smnzlnsk/mt-tum-artifacts-telemetry-network-optimization`. (accessed: 07/14/2025).

[65] S. Zelenski. *Monitoring Manager*. URL: `https://github.com/smnzlnsk/monitoring-manager`. (accessed: 07/06/2025).

[66] S. Zelenski. *monitoring-proto-lib*. URL: `https://github.com/smnzlnsk/monitoring-proto-lib`. (accessed: 07/01/2025).

[67] S. Zelenski. *Routing Manager*. URL: `https://github.com/smnzlnsk/routing-manager`. (accessed: 07/06/2025).