GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Evaluation of an Alternative System Monitoring Architecture for HPC Clusters

Henrik Jonathan Seeliger

Matrikelnummer: 20534843

Supervisor: Aasish Kumar Sharma

Georg-August-Universität Göttingen
Institute of Computer Science

March 16, 2025

Abstract

This project proposes and qualitatively evaluates an alternative system monitoring architecture for high-performance computing environments. The overall goal of this project is to offer an improvement to the monitoring process currently implemented at the Scientific Compute Cluster at the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen.

The proposed alternative architecture utilizes the OpenTelemetry project with its OpenTelemetry Collector agent, as well as the ClickHouse DBMS, to facilitate performant and flexible monitoring of distributed systems. The evaluation shows that this alternative architecture exhibits significant scalability, flexibility, performance and feature advantages when compared to the currently implemented architecture at the cost of increased resource requirements regarding the DBMS and the monitored applications.

Repository

The source code of the accompanying infrastructures, as well as the source of this document, can be accessed using the following URL:

https://gitlab.gwdg.de/h.seeliger/monitoring-system-performance

Declaration of Authenticity

Declaration on the Use of ChatGPT and Comparable Tools in the Context of Examinations

In this work I have used ChatGPT or another AI as follows:

- ☑ Not at all
- ☐ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Figures

List of Listings

Acronyms

API – application programming interface

DBMS – database management system

GWDG – Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

HPC – high-performance computing

KVM – Kernel-based Virtual Machine

OLAP – online analytical processing

OpAMP – Open Agent Management Protocol

OS – operating system

OTel – OpenTelemetry

OTLP – OpenTelemetry Protocol

QEMU – Quick Emulator

SCC – Scientific Compute Cluster

SDK – software development kit

SIMD – Single Instruction, Multiple Data

VM – virtual machine

1 Introduction

1.1 Motivation

The management of large-scale computing systems introduces a variety of challenges. One such challenge is to ensure the uninterrupted operation of the systems with its services. With an increasing number of distributed computers and components, this may become more difficult, as identifying the source and location of failures may need an increasing amount of time. With a significant number of running components, especially areas like cloud computing or high-performance computing (HPC) are presented with this challenge.

To mitigate this challenge, the process of monitoring can be employed. By gathering and analyzing system and application data, monitoring enables insights into the systems to identify the existence and causes of various issues. Therefore, the activity of monitoring plays an essential role in maintaining and improving the reliability and performance of computing systems, especially in the context of distributed and HPC systems [1].

However, choosing and implementing an monitoring architecture is not a trivial task and depends on various factors, such as given requirements. For HPC, various requirements exist, such as performance, storage efficiency and scalability [2]. The Scientific Compute Cluster (SCC) of the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG), for example, employs a monitoring architecture based on the software *Grafana*, *Telegraf* and the time-series database *InfluxDB* [3]. This architecture, often referred to as *TIG stack*, is considered to be an highly popular monitoring architecture [4].

Still, various other monitoring architectures exist. These may employ different components for, e.g., data storage and analysis, and may offer various advantages when compared to the currently implemented architecture. Other time-series databases have already been the subject of quantitative analysis, which has demonstrated performance and storage efficiency advantages for monitoring use cases in comparison to InfluxDB [5]. However, a qualitative comparison of the *TIG stack* and other possible monitoring architectures is still to be conducted.

1.2 Objectives

This project aims to explore and propose an alternative system monitoring architecture as a potential enhancement to the current monitoring architecture implemented in the SCC. This architecture can then be evaluated by comparing it to the current architecture, highlighting its qualitative advantages and disadvantages.

To achieve this, possible alternative architectures and technologies are researched. These are then used to develop and implement an alternative architecture by creating a reproducible

virtual machine (VM) containing an example instance of the new architecture. This VM enables the exploration of the new architecture and the analysis of its characteristics. A comparable VM for the currently employed architecture, the *TIG stack*, is implemented as well. These VMs can then be explored and analysed, thereby serving as the foundation for the comparison and evaluation. After comparing both architectures, advantages and disadvantages of both architectures can be stated and the new architecture can be evaluated for its utilization in HPC systems.

Thereby, the primary contribution of this project is represented by an alternative system monitoring architecture, accompanied by an analysis and evaluation of its characteristics. In addition, a reproducible baseline in the form of VMs for comparing infrastructures and system architectures is presented, allowing further comparative evaluations in the future.

The qualitative evaluation conducted in this project shows performance, scalability, flexibility and feature advantages of the alternative architecture when compared to the currently implemented architecture. These benefits, however, come at the cost of increased resource requirements regarding the database and, depending on the utilized approach, regarding the monitored applications.

1.3 Outline

This report is structured as follows. Section 2 presents background information regarding system monitoring and the currently utilized monitoring architecture. Subsequently, the methodology employed in this project is described briefly in Section 3. The results of this project are presented in Section 4, namely the alternative architecture and its evaluation. Section 5 then presents the discussion of challenges encountered during this project and possible alternatives. The report is then concluded in Section 6.

2 Background

This section introduces the concept of system monitoring, as well as the monitoring architecture currently implemented in the SCC. Furthermore, the related work addressed in Section 1.2 and its significance for this project are explained.

2.1 System Monitoring

System monitoring can be defined as the technologies and processes to measure various aspects of computer systems. The information gained by this procedure can be used to identify problems and increase the system's performance and reliability [6], [7]. The monitored aspects of systems may not only include system-level information such as CPU and RAM usage, but also data about running services. This data may be of different nature and includes, e.g., metrics and logs. Section 4.1 describes a formal definition of these concepts.

Various approaches to facilitate the process of system monitoring exist [8]. In the majority of cases, architectures are divided into multiple stages, with each stage being assigned a different task using different technologies. Typically, the persistence stage stores the data for preservation and further analysis. This can be realized using various database management systems. Due to the temporal nature of the data, a time-series database is particularly suited. The collection stage is then able to gather data from a configurable range of sources and store it using the persistence stage. Using a graphical user interface, e.g., the collected data can be visualized and explored, as implemented by the visualization stage. This high-level architecture can serve as the base for monitoring systems, while other architectures with different approaches tailored to specific requirements exist [1], [2], [8], [9].

Each stage requires the consideration of various factors. Besides different approaches for each stage, a range of technologies exist [8]. The choice of specific approaches and components depends on the specific requirements to the given environment. Monitoring in the context of HPC can be considered to be closely related to monitoring in the context of cloud computing, as both fields share architectural characteristics and objectives. For this field, a number of requirements exist, which can be transferred to monitoring in the context of HPC. These requirements include *scalability*, *timeliness* and *extensibility* [10]. As for the field of HPC and the given reference environment of the SCC, *performance efficiency* and a near real-time timeliness of data collection and analysis are given additionally [3]. However, one difference between the requirements for cloud computing and HPC is an increased significance of the monitoring system's performance and resource efficiency for the field of HPC, as their impact on the HPC systems should be minified.

2.2 Current Architecture

The reference architecture implemented in the SCC, which represents the basis for the evaluation, employs the three stages described in the previous section. To implement the persistence stage, the time-series database *InfluxDB* [11] is utilized. On each server, the *Telegraf* [12] agent is employed to collect the local monitoring data and send it to the database, thereby forming the collection stage. For visualization and exploration purposes, the SCC utilizes the *Grafana* [13] application, which queries the monitoring data from the InfluxDB database. In this project, version 2 of InfluxDB is assumed to be utilized. Figure 1 illustrates the current architecture.

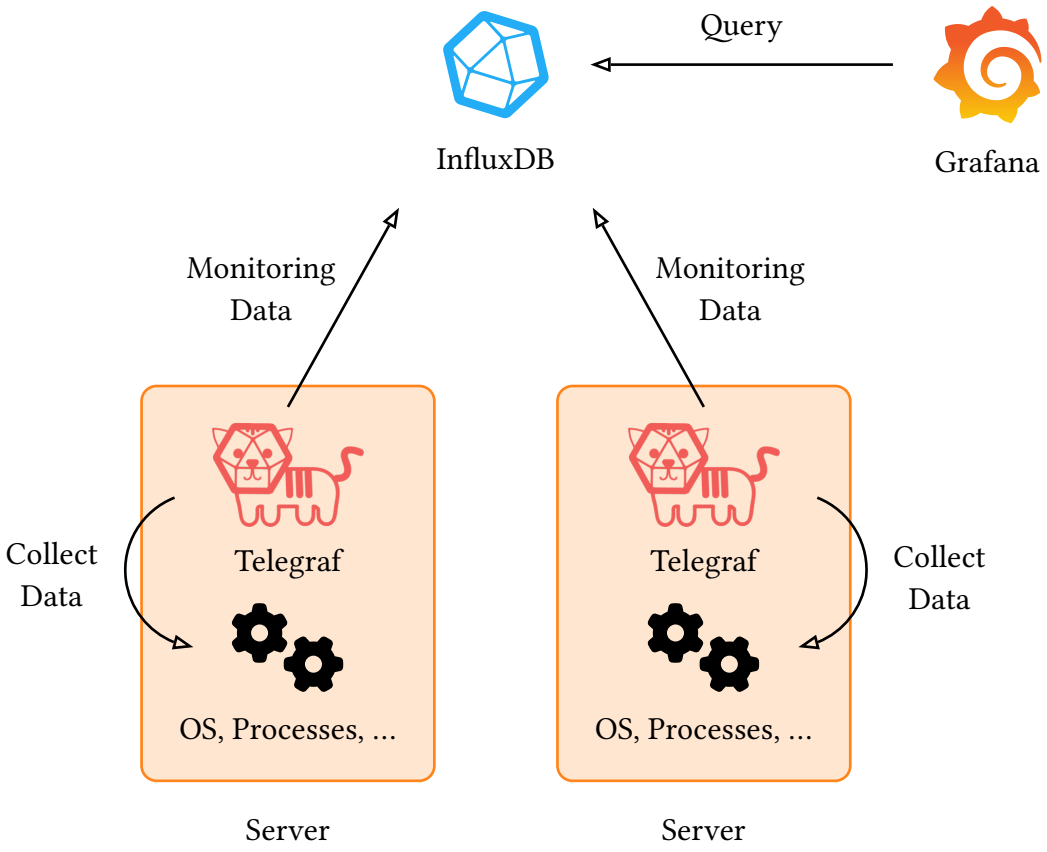

Figure 1: Currently employed monitoring architecture in the SCC.
Icons taken from [12], [13], [14], [15].

This architecture can be characterized as *agent-based* and *active*. Agent-based architectures employ software *agents* that collect monitoring data and send it to the persistence stage. As the data is actively collected by Telegraf instances, the architecture is considered to be *active*. Conversely, in a *passive* architecture, the monitored resources can send their data to the agent themselves. The omission of the agents would result in an *agent-less* architecture. Further technical characteristics of the architecture are described and discussed in Section 4.

2.3 Related Work

The publication *TSM-Bench: Benchmarking Time Series Database Systems for Monitoring Applications* by Khelifati et al. offers a comprehensive method for benchmarking time series databases in the context of monitoring [5]. The measured characteristics include performance as well as storage efficiency. In this publication, the method has been applied to various existing time-series databases, including InfluxDB 1.7.10-1 with InfluxQL as utilized query language. In numerous experiments, InfluxDB demonstrated poor performance and storage efficiency when compared to other time-series databases. Specifically, InfluxDB only performed adequately in data fetching queries involving low-volume data. However, with an increased data volume, InfluxDB exhibited a significant decline in query performance and storage efficiency. In addition to quantitative aspects, InfluxDB showed another disadvantage, namely the inability to test certain scenarios due to missing support for advanced combinations of aggregate functions.

These aspects provide initial insights for the comparison and serve as the foundation for the motivation of this work. However, it is important to note that the benchmarked version of InfluxDB is obsolete. The most recent available open-source version of InfluxDB is InfluxDB 2, which has introduced a new query language, *Flux*, and deprecated InfluxQL [16]. This project does not cover a quantitative comparison, e.g., using benchmarking. Still, a more detailed comparison would benefit from an updated benchmark and should be pursued in the future.

3 Methodology

The objective of this project is to explore and propose a potential improvement to the system monitoring architecture currently implemented in the SCC. The methodology to accomplish this objective is as follows.

The suitability of technologies and solutions for a given problem depends on the specific requirements and the degree to which these requirements are met. Therefore, research must be conducted regarding the requirements for system monitoring in the context of HPC and the SCC. Subsequent to the collection of requirements, potential technologies and aspects of the improved architecture can be researched, with the aim of fulfilling these requirements. The knowledge acquired regarding requirements and technologies can then be employed to develop and implement an alternative system monitoring architecture. The result of the implementation should be a reproducible and functional instance of this architecture to facilitate practical exploration and evaluation. Additionally, the reference architecture should be implemented in the same way for the comparison. Both instances should be realized using declaratively configured VMs, using *Vagrant*. Vagrant is an open-source software tool to configure, provision and run local VMs using *Vagrantfiles* [17]. This approach ensures reproducibility and simplifies the setup of the environment. These VMs can then be utilized to explore both architectures practically in order to discover their various characteristics as a foundation for comparison and evaluation. After comparing both architectures, advantages and disadvantages of both the alternative and reference architecture can be highlighted, emphasizing the suitability of the alternative architecture.

The primary desired outcome of this project is the description and implementation of an alternative system monitoring architecture that offers numerous advantages when compared to the original architecture. This outcome should include a comprehensive overview of both architectures, with their characteristics, advantages, and disadvantages.

4 Results

This section describes the system monitoring architecture implemented in this project. Furthermore, characteristics of both the currently implemented and the alternative architecture are explained and opposed. This enables the analysis of benefits and drawbacks of both architectures, as presented by the subsequent evaluation.

4.1 Alternative Architecture

As described in Section 2.1, various requirements for system monitoring in the context of HPC have been identified from relevant literature, including scalability, performance efficiency and extensibility. Based on these requirements, technologies and methods for system monitoring architectures have been researched. These can be employed to design and implement the alternative system monitoring architecture.

First, a high-level structure of the architecture must be considered. This aspect of the alternative architecture describes the topology and process of the overall system monitoring and data flow, i.e., the structure and flow of the data from ingestion to analysis. After that, technological components for the various tasks in the architecture can be chosen. The following sections explain the different aspects and chosen components of the alternative architecture.

The visualization stage of the alternative architecture is implemented analogously to the one of the architecture currently implemented. It utilizes the *Grafana* application to offer visualizations and the possibility to explore and analyze the data stored in the persistence stage. Therefore, it is not presented separately.

4.1.1 Structure

Various publications describe different high-level structures for system monitoring architectures [8], [10]. As described in Section 2, a multi-stage approach represents one popular choice. This multi-stage approach is also employed by the current implementation of the SCC, thus it is selected as the foundation for the alternative architecture. This also simplifies the comparison process.

As with the current architecture, the alternative architecture is divided into three stages: collection, persistence, and visualization. The collection stage involves the resources to be monitored, such as system components and the applications running on these components, as well as collection agents. Analogous to the Telegraf services in the current architecture, the collection agents serve as decoupled components which forward the monitoring data to a specified destination. In the alternative architecture, the persistence stage, and therefore the monitoring database, represents this destination. Therefore, this alternative architecture can

be described as *agent-based*. The persistence stage can then be queried and analyzed by the visualization stage using a visualization application.

In contrast to the current architecture, however, the collection agents offer multiple methods for obtaining monitoring data. In cloud computing environments, the *active* approach of collecting monitoring has gained wide application. Many applications implement HTTP endpoints that collection agents can request to obtain the current monitoring data for the application. This approach is also known as *pull-based* approach, as the agent actively pulls the data from the resources [18]. The alternative architecture aims to additionally support the push-based approach, i.e., the resources such as applications push their monitoring data to the collection agents themselves. In these two approaches, the collection agents do not need to be deployed on the same physical components as the resources. However, in a third supported method of data collection, the collection agent is deployed directly on the system's components, e.g., physical servers, and collects the data directly from them. This is the method currently implemented in the SCC. As discussed later in Section 4.3, each approach has respective advantages and disadvantages, while supporting multiple approaches increases the architecture's flexibility. Figure 2 illustrates the alternative architecture's structure with its three data collection methods.

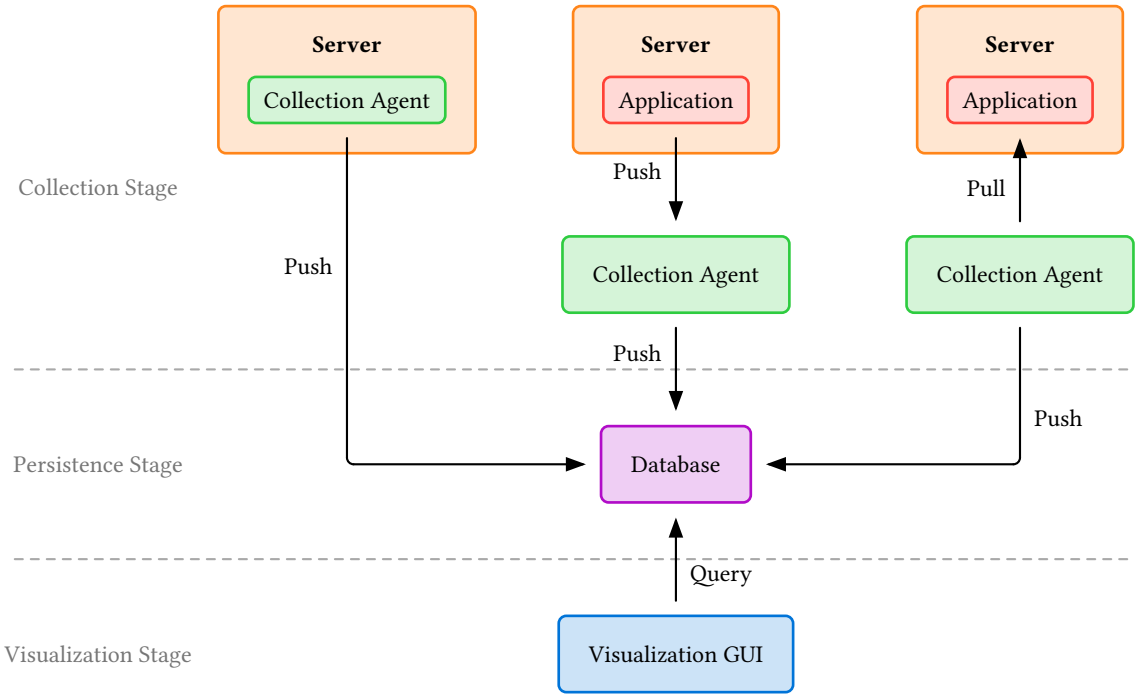

Figure 2: High-level view of the architecture with three different collection modes.

4.1.2 ClickHouse

In order to persist and analyze the monitoring data, the persistence stage utilizes a database management system (DBMS). The selection of the employed DBMS must be based on the requirements. A wide range of possible components exist that may meet these requirements. In this project, *ClickHouse* is chosen as DBMS.

ClickHouse is an open-source, column-oriented DBMS, focused on real-time and performant analytical queries [19]. Since its open-source release in 2016 by Yandex, it gained

popularity for online analytical processing (OLAP) use cases and has been adopted by companies such as Cloudflare and Uber [20]. For data manipulation and analytics, it employs the relational data model and offers a query language based on SQL with a wide range of advanced functionality when compared to the ANSI SQL standard. This functionality includes support for time-series data with various analytical functions.

According to several sources, ClickHouse exhibits a significant performance advantage for analytical queries in comparison to other DBMSs, particularly in the context of monitoring use cases, as described in Section 2.3 [5], [20], [21]. Together with its capacity of advanced analytical queries on temporal data, this characteristic represents the primary reason for selecting it as the alternative architecture's persistence stage.

4.1.3 OpenTelemetry

Many open-source projects in the field of monitoring work with custom nomenclatures and semantics for monitoring data. For example, the open-source project *Prometheus*, which acts as a collection agent and monitoring database simultaneously, defines specific protocols and semantics that monitored applications have to implement [18]. To be monitored by Prometheus, applications have to expose an HTTP endpoint which can be scraped by Prometheus according to its protocol and semantics.

While many applications adopted the monitoring model of Prometheus, the approach of implementing vendor-specific monitoring methods exhibits various limitations. One limitation is that the resulting monitoring system is coupled to the format and methods of the vendor, limiting its usability and extensibility. Additionally, integration and migration to other vendors may prove to be complicated and prone to errors as no common formats or semantics exist.

These aspects, among others, serve as the motivation for the OpenTelemetry (OTel) project. OTel is a vendor-neutral framework for observability, which standardizes semantics, protocols, formats and application programming interfaces (APIs) for monitoring [22]. In contrast to the limited scope of monitoring projects that typically emphasize a single monitoring data type, such as metrics as observed in Prometheus, OTel offers an unified approach by encompassing multiple data types. Besides metrics, logs and traces are supported. The protocol for monitoring data transmission defined by OTel, the OpenTelemetry Protocol (OTLP), utilizes either HTTP or gRPC and facilitates a push-based approach where monitored applications push their monitoring data using this protocol to consumers such as collection agents. While OTel defines APIs for applications to integrate monitoring into them, several implementations for these APIs for various programming languages exist [23]. These implementations enable a simplified setup of monitoring for applications. Additionally, semantics and tools exist to convert existing monitoring formats such as the Prometheus format to the OTel model. Because of these aspects, OTel serves as the conceptual and practical base for the alternative architecture. Primarily, this is facilitated by employing the OTel Collector as the architecture's collection agents.

The OTel Collector is an observability data collection agent, comparable to Telegraf in the currently implemented architecture [24]. The Collector, which is based on the OTel specifications, provides various methods for acquiring monitoring data. These include the OTLP and support for additional protocols such as the Prometheus protocol. Furthermore, it is capable of writing the different monitoring data to a ClickHouse instance. The combination of these

features, in conjunction with its integration within the OTel framework, serves as the primary factor in its selection as the architecture's collection agent.

4.2 Implementation

To facilitate the evaluation, instances of both architectures to compare are implemented using reproducible VMs. Using the open-source tool *Vagrant*, both architectures can be instantiated in an automated and reproducible manner [17]. Subsequently, these instances can be explored and analyzed practically.

Both VMs are described using *Vagrantfiles*. A Vagrantfile determines the configuration of a VM, such as resources and initialization scripts. Using this file, Vagrant can automatically initialize the VM, using a configurable virtualization backend. In this project, the *libvirt* backend is chosen, utilizing Kernel-based Virtual Machine (KVM)/Quick Emulator (QEMU) as hypervisor, as these enable near-native performance [25]. Listing B.1 shows the Vagrantfile for the alternative monitoring architecture.

The SCC primarily utilizes the OS *Rocky Linux 8* on its computing nodes [26]. Therefore, the VMs are also based on this OS. To facilitate a comparison and evaluation, both instances implement the same monitoring use case. Both instances provide the persistence and visualization stage of their respective architecture, while their collection stage collects two types of metric data. First, they collect host-level metrics themselves, such as CPU and RAM utilization. Second, they scrape a given application using the Prometheus protocol to enable insights regarding both architecture's interoperability. As both architectures already deploy the Grafana application, which exposes a Prometheus-compatible endpoint for obtaining its metrics, this application is scraped in both instances. Additionally, the alternative architecture gathers logs from *journald*, which represents the logging system of the most widely adopted Linux init system *systemd* [27]. At the present time, Telegraf lacks the capacity to gather journald logs, due to its semantic model not being designed for logs, as opposed to OTel and its Collector [28].

4.3 Qualitative Evaluation

After specifying the alternative architecture and implementing instances for each, an exploration and comparison of the two architectures is possible. The results of the comparison can then be explained and subsequently evaluated to highlight the benefits and drawbacks of each architecture.

4.3.1 Comparison

4.3.1.1 Structure Level

Both architectures exhibit a number of similarities in their overall structure. They both utilize a staged approach, employing three stages for monitoring and an equal visualization stage. Additionally, both architectures implement an agent-based approach, in which agents transfer the obtained monitoring data to the persistence stage.

The agent-based approach and the integration of Telegraf and the OTel Collector enable configurable architecture instances. Both Telegraf and the OTel Collector offer a wide range of components for obtaining, processing and transmitting monitoring data. These components include the support for a variety of storage backends such as different DBMSs. Additionally, both agents offer transformation operations, enabling pre-processing operations of monitoring data, such as filtering and data cleaning. When combined, the configurable components enable flexible and complex data pipelines in the case of each agent technology.

By supporting a wide range of components for obtaining monitoring data, both collection agents allow for multiple data collection methods. Although not configured and implemented in the current architecture, both agent technologies support pull- as well as push-based approaches for collecting data. This includes the support for the OTLP in both cases.

However, both architectures show a number of differences at the structural level. The alternative architecture is based on the OTel framework, which provides a formal standard for formats, transmission, and semantics of monitoring data. The alternative architecture follows this standard. As the OTel standard uniformly supports metrics, traces, and logs, the alternative architecture implements these monitoring data types as well, while the currently implemented architecture is focused solely on metrics.

As the alternative architecture is based on this standard, it also follows the passive, push-based approach preferred by OTel and the OTLP. Primarily, applications push their monitoring data to the OTel Collector instances themselves, while active approaches, e.g., the approach of sending requests to Prometheus-compatible endpoints of the applications, are also supported. The current architecture only focuses on the active, push-based approach.

Another difference are the methods of configuring the collection agents. Both architectures support the declarative, file-based configuration of the agents. However, Telegraf, which is employed by the current architecture, is configured using INI files, while the OTel Collector is configured using YAML files. Both configuration approaches enable the definition of complex monitoring data pipelines. Additionally, the OTel Collector supports remote configuration using a standardized remote management protocol, named Open Agent Management Protocol (OpAMP). Listing B.4 and Listing B.5 show the configuration files for both collection agents in this project.

4.3.1.2 Data Storage and Analysis Level

For storing and analyzing the monitoring data, both architectures employ a persistence stage. In this stage, the monitoring data is persisted using a DBMS, which additionally offers data analysis capabilities using query languages. While the current architecture utilizes InfluxDB, the alternative architecture implements its persistence stage using ClickHouse.

When comparing both persistence stages and their different DBMSs, only a limited number of similarities show. One similarity is that both databases support time-series data, which is essential for supporting the use case of monitoring data and its temporal nature. Being a time-series database, InfluxDB focuses on temporal data, while ClickHouse is based on the relational data model with a wide range of supported data types. Still, these data types include time and time-dependant data. Another similarity is represented by the capability of both databases to offer extensive languages and tools for querying and analyzing the stored data.

The employed languages for querying and analyzing data, however, present the first difference between both architectures and their utilized DBMSs. The current open-source version

of InfluxDB, InfluxDB 2, provides the functional data scripting language *Flux* as primary query language. This language has been specifically developed for InfluxDB and offers a wide range of functionalities for manipulating and processing time-series data. Additionally, InfluxDB 2 provides limited support for InfluxQL, a SQL-like language that was developed for InfluxDB and for managing time-series data. This language, however, is less expressive than, e.g., the ANSI SQL standard, as it does not support various standardized features such as *joins*. Furthermore, a number of restrictions apply to the utilization of InfluxQL in InfluxDB 2 due to its deprecation, such as missing support for *ALTER* and *CREATE* statements. One important aspect, however, is that with the announcement of InfluxDB 3 in September 2023, both InfluxQL and Flux have been deprecated [29]. InfluxDB 3 introduces a new SQL-based language, but at the present time, no stable open-source release of InfluxDB 3 exists [30].

ClickHouse also supports a SQL-like language, *ClickHouse SQL*, which extends the ANSI SQL standard with many advanced analytical functions. Both Flux and ClickHouse SQL offer a variety of functions, e.g., for processing time-series and semi-structured data, including array and dictionary data types. However, ClickHouse SQL additionally supports more advanced analysis functions such as machine learning aggregation functions and time-series-related algorithms. Listing B.2 and Listing B.3 show the same query for calculating the current CPU utilization implemented using Flux and ClickHouse SQL. This provides a simplified overview of the languages and their functions. However, a more in-depth investigation of both languages is beyond the scope of this project.

The data storage implementations of both database systems represent another difference. For persisting data, InfluxDB offers one storage engine. ClickHouse, on the other hand, offers over 22 table engines with different underlying technologies and methods. These include table engines optimized for specific aggregation use cases such as *Summing Merge Trees*, as well as special engines for, e.g., distributed and memory-only tables. Besides the possibility of changing the underlying storage engine, ClickHouse offers additional configuration options for optimizing performance and storage efficiency, such as configurable compression and partitioning. Furthermore, ClickHouse supports horizontal scaling for data replication, while InfluxDB offers clustering capabilities only in its commercial version.

A notable distinction between the two DBMSs lies in their performance characteristics. ClickHouse was designed for OLAP use cases and is thereby developed for the fast processing and analysis of datasets. To facilitate this, various techniques have been implemented, such as computation parallelization using Single Instruction, Multiple Data (SIMD) and multi-node processing [19]. A number of benchmarks exist which demonstrate a significant performance advantage of ClickHouse when compared to other DBMSs, including InfluxDB [21], [31]. These benchmarks also investigate time-series use cases. For monitoring use cases, Khelifati et al. show a significant performance advantage of ClickHouse when compared to InfluxDB regarding query performance and storage efficiency [5].

4.3.2 Benefits and Drawbacks

In the preceding section, a variety of characteristics for each architectural design is explained and compared. These characteristics can now be utilized to derive the benefits and drawbacks of the alternative architecture for the context of HPC.

The first aspect to consider is the passive, push-based approach of the alternative architecture in the context of application monitoring. In this approach, the applications send their

monitoring data to the collection agents themselves. However, this may have a negative impact on the performance of the monitored applications, as they are required to implement and execute logic for the collection and transmission of the data. Tracing, e.g., requires the application to record the timing of internal functions calls, creating an overhead for the application. In contrast, the passive approach enabled by OTel simplifies the configuration of application monitoring and enhances the alternative architecture's scalability. By providing software development kits (SDKs) as well as auto instrumentation capabilities for a variety of programming languages and environments, applications may be integrated into the monitoring system in a simplified manner. These applications can subsequently be configured only using environment variables, as required by the OTel API. This simplified configuration enhances the overall architecture's scalability, as new applications can be integrated with reduced effort. Furthermore, the passive, push-based approach of OTel offers an increased scalability when opposed to the active approach of the current architecture by relying on a stateless, unidirectional communication and flow of monitoring data. As the OTel Collector and the OTLP can be considered push-based and stateless, the flow of the data and thereby the architecture can be simplified while increasing its flexibility. Applications and OTel Collector instances only have to be configured once to send or, in the case of the Collectors, send and receive OTel data. The active approach, however, requires a reconfiguration of the agents to add new applications to monitor. Additionally, the scraping logic has to be synchronized and distributed among the agents to prevent duplicated data and performance degradations.

Due to the performance implications associated with the passive approach of the alternative architecture, this approach is only applicable for applications and environments with less restrictive performance requirements. In cloud computing environments, the performance impact of applications sending their monitoring data may be seen as negligible, because the significance of reliability and therefore of monitoring might outweigh the significance of performance. However, the context of HPC is defined by strict performance requirements, rendering the passive approach as not applicable in the majority of HPC environments. Still, given the scalability and flexibility advantages described previously, this approach may be utilized for applications with less strict performance requirements, such as job schedulers. Additionally, depending on the application's requirements, OTel enables applications to be compiled with monitoring capabilities using the OTel SDKs in a debug configuration. In a test run, metrics, traces and log data may then be collected and analyzed using the monitoring system to optimize the performance of the application. For a production run and its intended purpose, the application can be built without the monitoring capabilities to ensure optimal performance.

The previously described aspects represent scalability, flexibility and observability benefits of the alternative architecture in the context of monitoring applications. The benefits of scalability can be transferred to the monitoring of the systems as well. In the context of monitoring the systems directly, both architectures utilize an equal approach of agents being deployed on the systems and actively collecting host metrics. Here, the OTel Collector and the alternative architecture support metrics, traces and logs natively. This can be utilized to monitor additionally, e.g., the logs of the systems. Another benefit of the alternative architecture is the configurability of the OTel Collector and, thereby, the monitoring system. Both Telegraf and the OTel Collector support the configuration of complex data pipelines, including filtering and processing of the data before sending it to the persistence stage. The nature of INI files,

however, complicate this in the case of Telegraf. The configuration of the OTel Collector is strictly divided into receivers, processors, exporters, and the combination of these as pipelines. In contrast, Telegraf pipelines are configured at the component level, resulting in complex configuration files. Listing B.6 shows a Telegraf configuration with metric filtering enabled and illustrates, when compared to the OTel Collector configuration depicted in Listing B.5, this complexity.

The final aspects to consider for the evaluation of the alternative architecture concern their persistence stages, i.e., ClickHouse and InfluxDB. The query language of ClickHouse, a SQL-like language based on ANSI SQL, offers a number of benefits when compared to InfluxDB and its employed language Flux. Utilizing a query language that is based on one of the most widely known languages increases the architecture's adaptability for operators and developers [32]. In addition, ClickHouse SQL offers a variety of advanced functionalities for the analysis of time-series data and, consequently, monitoring data. Flux also provides analytical functions for time-series data. However, as ClickHouse is based on the relational data model, it supports more analytical use cases involving data that is different from time-series data. Logging and tracing represent two such use cases that may benefit from this aspect, as analyzing this data may require the relating of it to other data of different nature. The query language and data model of ClickHouse facilitate this process, as well as the storage of all monitoring data types in one place.

Another benefit of utilizing ClickHouse when opposed to InfluxDB is a significant performance advantage of analytical queries for analyzing the monitoring data. This enhancement enables the ingestion of more data while enabling the exploration and analysis of existing data in less time. Additionally, it facilitates the execution of more complex queries, leading to further insights into the system and its services. Besides these performance advantages, ClickHouse offers several storage efficiency benefits. As described in the preceding section, various sources demonstrate that ClickHouse outperforms InfluxDB in terms of storage efficiency. Furthermore, the storage engine of ClickHouse tables can be configured and tuned to achieve an optimal storage size for persisted data, including configurable compression of the data. In contrast, InfluxDB does not offer the configuration of its storage engine or the compression of storage units. This advantage of ClickHouse can also be utilized to achieve a data architecture compromising multiple storage tiers, e.g., a hot and cold storage architecture.

However, ClickHouse exhibits an important drawback when compared to InfluxDB. As ClickHouse is designed for performant, analytical queries, it requires a considerable amount of resources to operate when compared to InfluxDB and other DBMSs, primarily considering CPU and RAM. For ClickHouse, the recommended amount of RAM to operate is 32 GB or more. The operation of ClickHouse in envionments with a lower amount of RAM is possible, but specific tuning is necessary [33]. In this project, the VM running ClickHouse was assigned 4 GB of RAM with no observable functional impact. However, in the event of ClickHouse reaching its RAM limit, the system may cease to function correctly. This may result in service outages or data loss during data collection. In contrast, InfluxDB requires a comparatively lower amount of resources, with 32 GB of RAM only being necessary for a substantial volume of writes per second (more than 250,000 writes per second) and stored data (more than 1,000,000 unique series) [34]. Therefore, the required resources represent a significant drawback of the alternative architecture and have to be considered.

5 Discussion

After presenting the challenges faced in this project, possible alternatives to the architecture described in this project are shortly discussed.

5.1 Challenges

The most significant challenge encountered in this project is the lack of resources on requirements for monitoring, specifically in the context of HPC. A limited amount of literature was identified that enables this work. However, to facilitate a more comprehensive comparative analysis, more requirements and use cases have to be investigated in the future. This could be achieved, for example, by surveys in fields related to HPC.

The quantity of resources available for monitoring architectures based on ClickHouse and the OTel Collector with its configuration was also restricted. A significant portion of alternative architecture had to be developed and configured by the author. Primarily, this includes the configuration of the various included technologies, such as ClickHouse and the OTel Collector. However, this facilitated the exploration of diverse architectures and enabled the learning of systems monitoring methodologies.

5.2 Alternatives

A wide range of alternative components and approaches for implementing monitoring architectures exist, which have not been covered in this project. These include alternative technologies for the collection agents and DBMSs. In the future, an examination of these approaches and technologies might prove as beneficial by providing additional improvements.

The developers of Grafana present *Grafana Alloy* as one alternative collection agent technology. This agent is based on the OTel Collector and adds various additional features such as clustering and a simplified distributed scraping of application metrics [35]. Furthermore, it employs a specific configuration language commonly utilized in the Grafana environment. As a part of the Grafana environment, it can be combined in a simplified manner with other Grafana products such as Grafana (the visualization application) or *Grafana Mimir*.

Released in 2022, *GreptimeDB* presents a relatively new DBMS with a focus on monitoring data [36]. This DBMS natively supports the ingestion of OTel data using the OTLP, while supporting multiple languages for querying and analyzing the data, such as SQL and *PromQL*, the language utilized by Prometheus. Because of its monitoring-focused feature set, this database may be investigated in the future to evaluate its employment in the persistence stage.

6 Conclusion

Ensuring the uninterrupted operation of large-scale computing systems poses a major challenge to HPC environments. The process of system monitoring mitigates this challenge by collecting data from running components, enabling the real-time analysis of the system and providing insights regarding its condition. Due to the number of available technologies and required components, a wide range of system monitoring architectures exists. The SCC, for example, utilizes the *TIG stack*. However, many alternative architectures exist which potentially exhibit advantages over the TIG stack and improve the monitoring process of systems such as the SCC.

The aim of this project was to explore and evaluate such an alternative system monitoring architecture to identify its benefits and drawbacks. With the findings of this comparison, system administrators may improve their monitoring process by employing aspects or components of the alternative architecture, depending on their requirements.

The analyzed alternative monitoring architecture is based on the OTel framework and utilizes the OTel Collector as well as the ClickHouse DBMS. The evaluation conducted in this project shows that the new architecture exhibits a number of advantages when compared to the currently employed TIG stack in terms of performance, scalability, flexibility and features. However, these benefits come at the cost of highly increased resource requirements for the persistence stage in terms of, e.g., required RAM, that may impact the monitored systems significantly. The alternative architecture offers multiple approaches for collecting monitoring data from applications and depending on the implemented approach, monitored applications may be affected additionally. For monitoring the systems directly, no impact of the new approach has been found. In addition to these findings, a framework for comparing system monitoring architectures is proposed that facilitates the practical and reproducible exploration of infrastructures.

Therefore, this project may be seen as successful. Besides the main findings described above, the author was able to research and learn a large number of infrastructure and monitoring aspects which can be utilized in future projects. One future project is presented by investigating the described shortcomings, such as benchmarks for the current versions of the architecture components. This may yield valuable insights and may improve system monitoring architectures in the future.

Bibliography

[1] E. Al-Shaer, H. Abdel-Wahab, and K. Maly, "HiFi: a new monitoring architecture for distributed systems management," in *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*, 1999, pp. 171–178. doi: 10.1109/ICDCS.1999.776518.

[2] S. Sanchez *et al.*, "Design and Implementation of a Scalable HPC Monitoring System," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1721–1725. doi: 10.1109/IPDPSW.2016.167.

[3] M. Merz, "Monitoring in High Performance Computing," 2024. Accessed: Dec. 12, 2024. [Online]. Available: https://hps.vi4io.org/_media/teaching/autumn_term_2024/hpcsa/monitoring.pdf

[4] InfluxData Inc., "Grafana Guide – A Guide to the TIG Stack – Telegraf, InfluxDB, and Grafana," 2025, Accessed: Jan. 30, 2025. [Online]. Available: https://www.influxdata.com/grafana/

[5] A. Khelifati, M. Khayati, A. Dignös, D. Difallah, and P. Cudré-Mauroux, "TSM-Bench: Benchmarking Time Series Database Systems for Monitoring Applications," *Proc. VLDB Endow.*, vol. 16, no. 11, pp. 3363–3376, Jul. 2023, doi: 10.14778/3611479.3611532.

[6] J. Turnbull, *The Art of Monitoring*. 2016.

[7] G. Wiesen, "What Is a System Monitor?," May 2024, Accessed: Jan. 31, 2025. [Online]. Available: https://www.easytechjunkie.com/what-is-a-system-monitor.htm

[8] J. M. Alcaraz Calero and J. Gutiérrez Aguado, "Comparative analysis of architectures for monitoring cloud computing infrastructures," *Future Generation Computer Systems*, vol. 47, pp. 16–30, 2015, doi: https://doi.org/10.1016/j.future.2014.12.008.

[9] R. Izadpanah, N. Naksinehaboon, J. Brandt, A. Gentile, and D. Dechev, "Integrating Low-latency Analysis into HPC System Monitoring," in *Proceedings of the 47th International Conference on Parallel Processing*, in ICPP '18. Eugene, OR, USA: Association for Computing Machinery, 2018. doi: 10.1145/3225058.3225086.

[10] G. Aceto, A. Botta, W. de Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013, doi: https://doi.org/10.1016/j.comnet.2013.04.001.

[11] InfluxData and Contributors, "InfluxDB." Accessed: Feb. 01, 2025. [Online]. Available: https://github.com/influxdata/influxdb

[12] InfluxData and Contributors, "Telegraf." Accessed: Feb. 01, 2025. [Online]. Available: https://github.com/influxdata/telegraf

[13] Grafana Labs and Contributors, "Grafana." Accessed: Feb. 01, 2025. [Online]. Available: https://github.com/grafana/grafana

[14] The Linux Foundation, "Cloud Native Landscape." Accessed: Feb. 01, 2025. [Online]. Available: https://landscape.cncf.io/

[15] Fonticons, Inc., "Server Icon." Accessed: Feb. 01, 2025. [Online]. Available: https://fontawesome.com/icons/server?s=solid

[16] R. Savage, "InfluxDB 2.0 Open Source is Generally Available," Nov. 2020, Accessed: Feb. 01, 2025. [Online]. Available: https://www.influxdata.com/blog/influxdb-2-0-open-source-is-generally-available/

[17] HashiCorp, Inc. and contributors, "Vagrant." Accessed: Feb. 03, 2025. [Online]. Available: https://github.com/hashicorp/vagrant

[18] J. Turnbull, "Monitoring with Prometheus." Turnbull Press, Jun. 12, 2018.

[19] R. Schulze, T. Schreiber, I. Yatsishin, R. Dahimene, and A. Milovidov, "ClickHouse – Lightning Fast Analytics for Everyone," *Proceedings of the VLDB Endowment*, vol. 17, pp. 3731–3744, Nov. 2024, doi: 10.14778/3685800.3685802.

[20] F. Lardinois, "ClickHouse launches ClickHouse Cloud, extends its Series B," *TechCrunch*, Dec. 2022, Accessed: Feb. 07, 2025. [Online]. Available: https://techcrunch.com/2022/12/06/clickhouse-launches-clickhouse-cloud-extends-its-series-b/

[21] A. Milovidov and contributors, "ClickBench: a Benchmark For Analytical Databases." Accessed: Feb. 07, 2025. [Online]. Available: https://github.com/ClickHouse/ClickBench/

[22] OpenTelemetry Authors, "OpenTelemetry Specification 1.41.0." 2024. Accessed: Feb. 08, 2025. [Online]. Available: https://opentelemetry.io/docs/specs/otel/

[23] OpenTelemetry Authors, "Language APIs & SDKs," 2024, Accessed: Feb. 08, 2025. [Online]. Available: https://opentelemetry.io/docs/languages/

[24] OpenTelemetry Authors, "Collector," 2024, Accessed: Feb. 08, 2025. [Online]. Available: https://opentelemetry.io/docs/collector/

[25] Red Hat, Inc., "Hypervisor im Vergleich: KVM oder VMware?," Mar. 2023, Accessed: Feb. 10, 2025. [Online]. Available: https://www.redhat.com/de/topics/virtualization/kvm-vs-vmware-comparison

[26] GWDG, "CPU Partitions," *GWDG HPC Documentation*, Feb. 2025, Accessed: Feb. 10, 2025. [Online]. Available: https://docs.hpc.gwdg.de/how_to_use/compute_partitions/cpu_partitions/index.html

[27] B. Byfield, "A Survey of Init Systems," May 2021, Accessed: Mar. 06, 2025. [Online]. Available: https://www.linux-magazine.com/Online/Features/A-Survey-of-Init-Systems

[28] M. Palmersheim, "Feature Request Systemd_Journald plugin." Accessed: Feb. 10, 2025. [Online]. Available: https://github.com/influxdata/telegraf/issues/8154

[29] InfluxData Inc., "InfluxData Unveils Future of Time Series Analytics with InfluxDB 3.0 Product Suite," Apr. 2023, Accessed: Feb. 11, 2025. [Online]. Available: https://www.influxdata.com/blog/influxdata-announces-influxdb-3-0/

[30] P. Dix, "The Plan for InfluxDB 3.0 Open Source," *InfluxDB Blog*, Sep. 2024, [Online]. Available: https://www.influxdata.com/blog/the-plan-for-influxdb-3-0-open-source/

[31] Altinity Team, "ClickHouse® Crushing Time Series," Nov. 2018, Accessed: Feb. 12, 2025. [Online]. Available: https://altinity.com/blog/clickhouse-for-time-series

[32] Stack Overflow, "Stack Overflow Developer Survey 2024 – Technology." Accessed: Feb. 14, 2025. [Online]. Available: https://survey.stackoverflow.co/2024/technology

[33] ClickHouse, Inc. and Contributors, "Usage Recommendations," *ClickHouse Documentation*, 2025, Accessed: Feb. 15, 2025. [Online]. Available: https://clickhouse.com/docs/en/operations/tips#ram

[34] InfluxData Inc. and Contributors, "Hardware sizing guidelines," *InfluxData Documentation*, 2023, Accessed: Feb. 15, 2025. [Online]. Available: https://docs.influxdata.com/influxdb/v1/guides/hardware_sizing/

[35] Grafana Labs and Contributors, "Grafana Alloy Documentation," 2025, Accessed: Feb. 15, 2025. [Online]. Available: https://grafana.com/docs/alloy/latest/

[36] Greptime and Contributors, "GreptimeDB OSS," 2025, Accessed: Feb. 15, 2025. [Online]. Available: https://greptime.com/product/db

[37] InfluxData and Contributors, "Configuration: Metric Filtering," *Telegraf Docs*, 2025, Accessed: Feb. 13, 2025. [Online]. Available: https://github.com/influxdata/telegraf/blob/master/docs/CONFIGURATION.md#metric-filtering

Appendix

A Figures

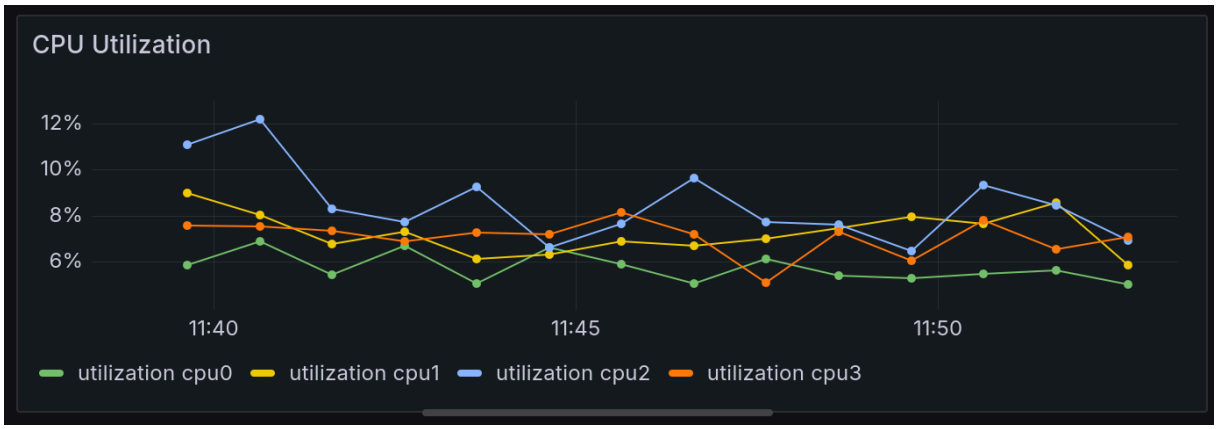

Figure A.1: Visualization of the result of the ClickHouse SQL query depicted in Listing B.2.

B Code Samples

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure("2") do |config|
  config.vm.box = "generic/rocky8"
  config.vm.box_check_update = false
  config.vm.hostname = "otc"

  config.vm.network "forwarded_port", guest: 8123, host: 3010
  config.vm.network "forwarded_port", guest: 3000, host: 3011

  config.vm.provider :libvirt do |libvirt|
    libvirt.cpus = 4
    libvirt.memory = 4096
    libvirt.machine_virtual_size = 20

    # See https://wiki.openstack.org/wiki/LibvirtXMLCPUModel
    libvirt.cpu_mode = "host-passthrough"
  end

    config.vm.provision "file", source: "./clickhouse/config.xml", destination: "/tmp/clickhouse.xml"
   config.vm.provision "file", source: "./clickhouse/monitoring_user.xml", destination: "/tmp/clickhouse_user.xml"
    config.vm.provision "file", source: "./otelcol/config.yaml", destination: "/tmp/
```

```
otelcol.yaml"
24    config.vm.provision "file", source: "./otelcol/dependencies.conf", destination: "/tmp/
otelcol-dependencies.conf"
25     config.vm.provision  "file",  source:  "./grafana/grafana.ini",  destination:  "/tmp/
grafana.ini"
26    config.vm.provision "file", source: "./grafana/datasource.yaml", destination: "/tmp/
clickhouse.yaml"
27
28   config.vm.provision "shell", inline: <<-'SHELL'
29     # Move configuration files to appropriate locations
30
31     mkdir -p \
32       /etc/clickhouse-server/config.d \
33       /etc/clickhouse-server/users.d \
34       /etc/otelcol-contrib \
35       /etc/grafana/provisioning/datasources \
36       /var/lib/grafana/plugins \
37       /etc/systemd/system/otelcol-contrib.service.d
38
39     mv /tmp/clickhouse.xml /etc/clickhouse-server/config.d/config.xml
40     mv /tmp/clickhouse_user.xml /etc/clickhouse-server/users.d/monitoring_user.xml
41     mv /tmp/otelcol.yaml /etc/otelcol-contrib/config.yaml
42          mv  /tmp/otelcol-dependencies.conf  /etc/systemd/system/otelcol-contrib.service.d/
dependencies.conf
43     mv /tmp/grafana.ini /etc/grafana/
44     mv /tmp/clickhouse.yaml /etc/grafana/provisioning/datasources/
45
46     # Install Grafana ClickHouse plugin
47
48     dnf install -y unzip
49
50     curl \
51       --fail --show-error \
52       -o /tmp/clickhouse.zip \
53          -L "https://grafana.com/api/plugins/grafana-clickhouse-datasource/versions/4.5.1/
download?os=linux&arch=amd64"
54     ls -al /tmp
55     unzip /tmp/clickhouse.zip -d /var/lib/grafana/plugins/
56     rm /tmp/clickhouse.zip
57
58     # Configure ClickHouse, OTEL Collector and Grafana repositories
59
60     dnf config-manager --add-repo https://packages.clickhouse.com/rpm/clickhouse.repo
61
62     cat <<EOF | tee /etc/yum.repos.d/grafana.repo
63 [grafana]
64 name=grafana
65 baseurl=https://rpm.grafana.com
66 repo_gpgcheck=1
67 enabled=1
68 gpgcheck=1
69 gpgkey=https://rpm.grafana.com/gpg.key
70 EOF
71
72     # Install packages
73
74     dnf install -y \
75       https://github.com/open-telemetry/opentelemetry-collector-releases/releases/download/
v0.115.1/otelcol-contrib_0.115.1_linux_amd64.rpm \
76       clickhouse-server \
77       clickhouse-client \
78       grafana
79
```

```
80      # Allow OpenTelemetry Collector to read journald logs
81
82      usermod -a -G systemd-journal otelcol-contrib
83
84      # Start services
85
86      systemctl daemon-reload
87      systemctl enable --now clickhouse-server otelcol-contrib grafana-server
88
89      # `otelcol-contrib` service is started automatically when installed,
90      # but as its user's groups were updated, it has to be restarted
91
92      systemctl restart otelcol-contrib
93    SHELL
94  end
95
```

Listing B.1: Vagrantfile for the comparison instance of the alternative architecture.

```
1  WITH cpu_times_per_state AS (
2    SELECT
3      time,
4      cpu,
5      CAST(
6        groupArray(
7          (state, value)
8        ),
9        'Map(String, Float64)'
10     ) AS states
11   FROM
12     (
13       SELECT
14         TimeUnix AS time,
15         Attributes[ 'cpu' ] AS cpu,
16         Attributes[ 'state' ] AS state,
17         Value AS value
18       FROM
19         otel_metrics_sum
20       WHERE
21         time >= $__fromTime
22         AND time <= $__toTime
23         AND MetricName = 'system.cpu.time'
24       ORDER BY
25         time ASC,
26         cpu ASC,
27         state ASC
28     )
29   GROUP BY
30     time,
31     cpu
32   ORDER BY
33     time ASC,
34     cpu ASC
35 ),
36 differences AS (
37   SELECT
38     time,
39     cpu,
40     lagInFrame(states) OVER (
41       PARTITION BY cpu
```

```
      ORDER BY
        time ASC ROWS BETWEEN UNBOUNDED PRECEDING
        AND UNBOUNDED FOLLOWING
    ) AS previous,
    states AS current,
    mapSubtract(current, previous) AS state_differences
  FROM
    cpu_times_per_state
  ORDER BY
    time ASC,
    cpu ASC
),
utilizations AS (
  SELECT
    time,
    cpu,
    state_differences[ 'interrupt' ]
    + state_differences[ 'nice' ]
    + state_differences[ 'softirq' ]
    + state_differences[ 'steal' ]
    + state_differences[ 'system' ]
    + state_differences[ 'user' ]
    + state_differences[ 'idle' ]
    + state_differences[ 'wait' ] AS total,
    state_differences[ 'interrupt' ]
    + state_differences[ 'nice' ]
    + state_differences[ 'softirq' ]
    + state_differences[ 'steal' ]
    + state_differences[ 'system' ]
    + state_differences[ 'user' ] AS active,
    active / total AS utilization
  FROM
    differences
  WHERE
    length(previous) > 0
)
SELECT
  time,
  cpu,
  utilization
FROM
  utilizations

```

Listing B.2: ClickHouse SQL query for calculating the current CPU utilization per CPU core in the alternative architecture.

Figure A.1 illustrates the result.

```
calculateUtilization = (usages) => {
    total = usages["time_nice"]
      + usages["time_softirq"]
      + usages["time_steal"]
      + usages["time_system"]
      + usages["time_user"]
      + usages["time_idle"]
      + usages["time_iowait"]

    active = usages["time_nice"]
      + usages["time_softirq"]
```

```
      + usages["time_steal"]
      + usages["time_system"]
      + usages["time_user"]

    return active / total * 100.0
}

from(bucket: "test-bucket")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "cpu")
  |> filter(fn: (r) => contains(
    value: r["_field"],
    set: [
      "time_nice",
      "time_softirq",
      "time_steal",
      "time_system",
      "time_user",
      "time_idle",
      "time_iowait"
    ]))
  |> filter(fn: (r) => contains(value: r["cpu"], set: ["cpu0", "cpu1", "cpu2", "cpu3"]))
  |> pivot(rowKey:["_time"], columnKey: ["_field"], valueColumn: "_value")
  |> derivative(
    unit: 1s,
    columns: [
      "time_nice",
      "time_softirq",
      "time_steal",
      "time_system",
      "time_user",
      "time_idle",
      "time_iowait"
    ])
  |> map(
      fn: (r) => ({
        _time: r._time,
        _measurement: r._measurement,
        _field: "cpu_utilization",
        _value: calculateUtilization(usages: r),
        cpu: r["cpu"]
      }),
    )

```

Listing B.3: Flux query for calculating the current CPU utilization per CPU core in the current architecture.

```
[agent]
  interval = "10s"
  metric_batch_size = 100

  flush_interval = "10s"

[[outputs.influxdb_v2]]
  urls = ["http://localhost:8086"]

  token = "test-token"
  organization = "test-org"
```

```
  bucket = "test-bucket"

[[inputs.cpu]]
  percpu = true
  totalcpu = true
  collect_cpu_time = true
  report_active = false
  core_tags = false

[[inputs.disk]]
  ignore_fs = ["tmpfs", "devtmpfs", "devfs", "iso9660", "overlay", "aufs", "squashfs"]

[[inputs.mem]]

[[inputs.opentelemetry]]

[[inputs.prometheus]]
  urls = ["http://localhost:3000/metrics"]

```

Listing B.4: Configuration of this project's Telegraf instance.

```
extensions:
  health_check:

receivers:
  hostmetrics:
    scrapers:
      cpu:
      disk:
      memory:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317
  prometheus:
    config:
      scrape_configs:
        - job_name: grafana
          scrape_interval: 10s
          static_configs:
            - targets: ["0.0.0.0:3000"]
  journald:
    directory: /run/log/journal
    units:
      - grafana-server
    priority: info

processors:
  batch:
    send_batch_size: 100
    timeout: 10s
  transform/set-hostmetrics-service:
    log_statements:
      - context: resource
        statements:
          - set(attributes["service.name"], "hostmetrics")
  transform/log-extract:
    log_statements:
      - context: log
```

```yaml
        statements:
          - set(severity_number, body["PRIORITY"])
          - set(attributes["unit"], body["_SYSTEMD_UNIT"])
          - set(attributes["syslog_identifier"], body["SYSLOG_IDENTIFIER"])
          - set(body, body["MESSAGE"])

exporters:
  clickhouse:
    endpoint: tcp://localhost:9000
    database: monitoring
    username: monitoring_user
    password: example_password
    create_schema: true

service:
  extensions: [health_check]
  pipelines:
    metrics:
      receivers: [otlp, prometheus]
      processors: [batch]
      exporters: [clickhouse]
    metrics/host:
      receivers: [hostmetrics]
      processors: [batch, transform/set-hostmetrics-service]
      exporters: [clickhouse]
    logs:
      receivers: [journald, otlp]
      processors: [batch, transform/log-extract]
      exporters: [clickhouse]

```

Listing B.5: Configuration of this project's OTel Collector instance.

```
[[inputs.cpu]]
  percpu = true
  totalcpu = false
  fieldexclude = ["cpu_time"]
  [inputs.cpu.tagdrop]
    cpu = [ "cpu6", "cpu7" ]

[[inputs.disk]]
  [inputs.disk.tagpass]
    fstype = [ "ext4", "xfs" ]
    path = [ "/opt", "/home*" ]

[[outputs.influxdb]]
  urls = [ "http://localhost:8086" ]
  database = "telegraf"
  namedrop = ["aerospike*"]

[[outputs.influxdb]]
urls = [ "http://localhost:8086" ]
database = "telegraf-aerospike-data"
namepass = ["aerospike*"]

[[outputs.influxdb]]
urls = [ "http://localhost:8086" ]
database = "telegraf-cpu0-data"
  [outputs.influxdb.tagpass]
```

```
27   cpu = ["cpu0"]
28
```

Listing B.6: Configuration example for Telegraf with filtering.
Taken from [37].