ANR SCALER

2023-2026

D 1.1

State of the Art on Microservice Performance Metrics

Chef de file pour le D 1.1 : Joaquim SOARES
Co-rédacteurs : Vania MARANGOZOVA

January 15, 2024

anr agence nationale de la recherche
UGA Université Grenoble Alpes
Inria
orange™

Contents

List of Figures

List of Tables

1 Introduction

In the dynamic landscape of modern software architecture, the advent of microservices has revolutionized the way applications are designed, developed, and deployed. The decomposition of monolithic structures into smaller, independently deployable units also known as *microservices* offers unprecedented agility and scalability. However, this architectural shift introduces a new set of challenges, foremost among them being the imperative to assess and optimize the performance of these distributed components.

Microservice applications are encapsulated into containers, a lightweight, portable, and efficient way to package, distribute, and run applications and their dependencies. Container orchestration platforms like are widely used to automate the deployment, scaling, and management of containerized applications. Kubernetes (K8S) [1] is widely used and has become a *de facto* standard in this landscape.

To ensure seamless functionality and user satisfaction, the role of performance metrics in microservices becomes paramount. These metrics serve as critical instruments, enabling users to systematically measure, analyze, and enhance the efficiency, reliability, and scalability of microservices within the Kubernetes ecosystem.

This report delves into the multifaceted objectives of performance metrics within the context of microservices, highlighting the intricate interplay between system behavior and the pursuit of optimal, responsive, and resilient distributed systems.

The report is organized as follows. Section 2 presents first-level performance metrics along with their goals. It also presents the default performance metrics management in K8S. Section 3 provides an overview of the approaches to collecting different performance metrics. The report continues with Section 4 which tackles the question of higher level performance indicators. The rich landscape of existing performance observation tools is presented in Section 5. Last, Section 6 concludes on the current state of art on performance metrics for microservices and points out the major challenges for efficient performance management.?

2 Microservice Performance Metrics

In the following we start with a description of the principal goals towards which microservice performance metrics are targeted (Section 2.1). We continue with an enumeration of the most used metrics according to the current state of the art (Section 2.2) and finish with a presentation of the monitoring architecture of Kubernetes (Section 2.3) which is the working context of the SCALER project.

2.1 Observation Goals

System observation involves an exploration of the system's behavior, interactions, and performance characteristics that contribute to a deeper understanding of the system dynamics. The metrics used to observe the behavior of a microservice-based system depend on the goal of observation. The main use cases include performance profiling, user experience, resources profiling, availability and scalability.

Performance Profiling. Profiling is a detailed analysis of individual service components to understand their computational, memory, storage and network resource utilization. It aims at identifying bottlenecks by providing insights into the runtime behavior of microservices. By leveraging metrics such as response times, throughput, and resource consumption, performance profiling facilitates targeted enhancements, ensuring that each microservice operates optimally within a larger system.

User Experience. Observation in this case focuses on the impact of service performance on end-users. It involves tracking metrics related to reliability and overall system responsiveness. By aligning observed data with user expectations, the goal is to ensure that the architecture delivers a satisfying and consistent experience. User experience metrics play a crucial role in assessing and enhancing the system, as they provide a direct link between the technical aspects of microservices and the value perceived by end-users.

Resource Usage Optimization. One of the major goals of observation is to check on the adequacy between resource allocation and demands and resource requirements, as well as on the resource usage and costs. It is to drive the efficiency of resource usage, minimizing contention, and improving overall system profitability. Resource allocation is a critical step that involves distributing compute resources among various service instances. Optimization in this context consists in fine-tuning these resources to enhance overall system efficiency. The aim is to minimize contention, identify unused resources, and ensure that each microservice receives the necessary resources for optimal performance. Moreover, the concept extends to financial considerations, emphasizing the cost-effectiveness of resource utilization. This entails assessing the balance between performance gains and infrastructure expenses. The main objective is to find an optimal trade-off where microservices operate efficiently without incurring unnecessary financial overheads.

Availability, Reliability and Fault Detection are paramount in a distributed context. Availability reflects the continuous operability of microservices, emphasizing minimal downtime and uninterrupted service delivery. Reliability extends this concept by focusing on correct functioning with consistent performance and the ability to meet user expectations over time. Fault detection is an integral part of maintaining availability and reliability, involving the proactive identification of anomalies and issues within microservices. This includes monitoring errors, exceptions, and deviations from expected behavior.

Scalability and Dynamic Adaptation are essential concepts, addressing the evolving nature of system demands. Scalability corresponds to an architecture's ability to efficiently manage and seamlessly adapt to variable workloads. Built on scalability, dynamic adaptation focuses on real-time adjustments to maintain optimal performance in a changing environment. This involves automated mechanisms that respond to fluctuations in workloads, resource availability, and other dynamic factors.

2.2 Performance Metrics

There are several recent review papers on microservice auto-scaling [2–5] which naturally tackle the question of *observation* and the underlying metrics. The metrics identified as the most widely used are the following:

- CPU and Memory. These are the main metrics used for characterizing the resource usage of microservices. They reflect the CPU usage and memory consumption. In the K8S ecosystem, the CPU usage is measured in *millicores* where $1core = 1000\ millicores$. Memory consumption is measured in the standard unit of *bytes* and its multiples (*KB, MB,GB*).
- Request Rate. This metric is usually used to quantify the load a microservice can absorb without degrading performance. The metric gives the number of requests served per second *req/s*.
- Response Time. Response time characterizes the time needed by a microservice to serve a request. The point of view is that of the client (user or another microservice) that has sent a

request. This metric is used to characterize the reactivity of a system (measured in *seconds* or its derivatives).

- Message Queue Length. Several works put forward the fact that the simple metrics of CPU and memory are not sufficient to detect system bottlenecks. They show that a more efficient metric comes from monitoring the lengths of message queues that come into play within microservice communications (interactions) [6–8] .
- Reaction time, Repair time, Recovery time, and Total Outage time are metrics that evaluate the availability of microservices after a scaling or a fault recovery has taken place.

Other important metrics are custom, higher-level, metrics and relate to the correct execution of microservices or to their execution cost. We identify the following:

- QoS Metrics. Quality-of-Service metrics may be considered from the application (semantics) point of view or from the system-side perspective. Concerning the latter, the standard performance evaluation corporation (SPEC) has investigated the question in the larger context of the cloud and not only for microservices [9]. They identify metrics related to the system's capacity for handling elasticity, availability, performance isolation, and operational risk. Related issues are reliability and System Level Objectives (SLO) [10], as well as Service Level Agreements (SLA) [11] .
- Infrastructure Cost (FinOps). As defined by the FinOps Foundation [12] "FinOps is a public cloud management discipline that enables organizations to get maximum business value from cloud by helping technology, finance, and business teams to collaborate on data-driven spending decisions.". Metrics including resource consumption, resources per server, energy consumption, cost per transaction, efficiency ratios, maintenance costs, etc. play a pivotal role for the good functioning of the cloud in general and microservices in particular.
- Accuracy Metrics. These are typically used in machine-learning-based systems wher ML models are to predict microservice performance or to take scaling decisions.

2.3 K8S Monitoring Architecture

According to K8S' documentation [13], it is possible to use either *resource metrics*, or *full metrics pipelines* to collect monitoring statistics.

Resource Metrics Pipeline The resource metrics pipeline provides a limited set of metrics collected by the lightweight, short-term, in-memory metrics-server [14]. These metrics are exposed via the `metrics.k8s.io API`. The standard monitoring K8S architecture that produces these metrics is shown in Figure 1.

The figure shows the controller node and two worker nodes in a K8S cluster. The API server which gives access to the collected metrics runs on the controller node.

On a worker node there is a special process, *kubelet* which is the primary node agent responsible of collecting end exposing pod and container metrics. The latter are made available either directly by *cAdvisor* which is intergated within kubelet, or via the the Container Runtime Interface (CRI) [16] based on gRPC [17]. Kubelet metrics include, among others, the number of running containers (`kubelet_running_container_count`)and the number of running pods (`kubelet_running_pod_count`). *cAdvisor* provides kubelet with information on the CPU (e.g `container_cpu_usage_seconds_total`), memory (e.g. `container_memory_working_set_bytes`), file system (e.g. `container_fs_usage_bytes`) and network (e.g. `container_network_receive_bytes_total`) usage.

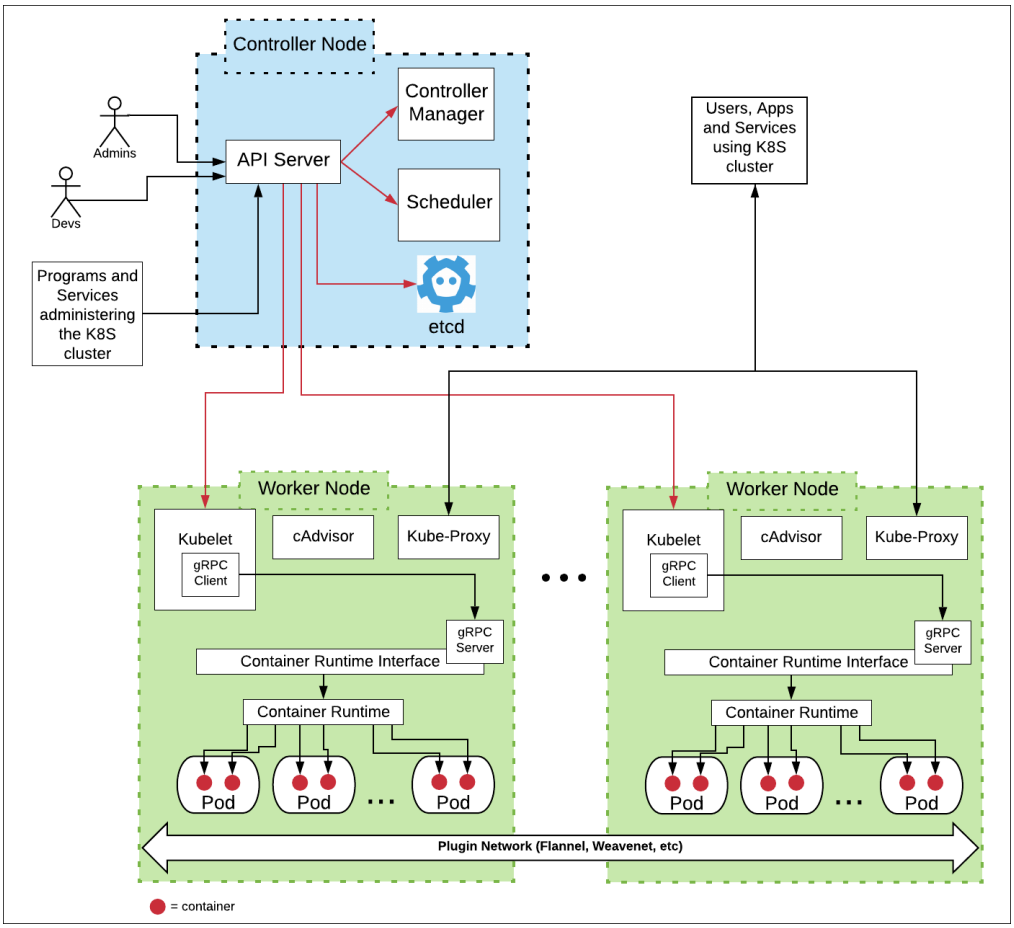

Figure 1: K8S Monitoring Architecture. Source [15].

Full Metrics Pipeline The definition of a full metrics pipeline is outside the scope of Kubernetes because of the very wide scope of possible solutions.

A widely-used monitoring tool in the K8S landscape is Prometheus [18] which is also a CNCF[1] open-source project. Prometheus accesses metrics via the *node exporter*[2]. The node exporter uses pluggable metric collectors among which the following are enabled by default:

```
cpu             Exposes CPU statistics
filesystem      Exposes filesystem statistics, such as disk space used.
meminfo         Exposes memory statistics.
netdev          Exposes network interface statistics such as bytes transferred.
rapl            Exposes various statistics from /sys/class/powercap
pressure        Exposes pressure stall statistics from /proc/pressure/
```

More information on the possible tools that may be plugged in a full pipeline is given in Section 5.

[1] Cloud Native Computing Foundation [**<empty citation>**]
[2] https://github.com/prometheus/node_exporter

3 Capturing Performance Metrics

In this section we lay the background on collecting performance metrics. We present four major aspects, namely instrumentation (Section 3.1), logging (Section 3.2), real-time monitoring (Section 3.3) and distributed tracing (Section 3.4).

3.1 Instrumentation

Definititon *Embedding observation capabilities in the microservice execution environment.*

Instrumentation is a fundamental concept in observation, representing the strategic embedding of observation capabilities (*probes*) directly into the codebase of individual microservices. The purpose is to collect detailed data on the internal workings of each service. This data often includes information about resource usage, execution times, and other performance-related metrics. By instrumenting code, developers and operators can strategically place probes to gain real-time insights into the behavior of microservices. major challenge is to prevent performance overhead.

The force of instrumentation lies in its ability to provide a fine-grained understanding of how microservices operate. It allows for the identification of performance bottlenecks and areas for optimization, facilitating data-driven decision-making in the pursuit of system efficiency.

Examples of instrumentation metrics :

- **Response Times** measure the time taken for microservices to respond to requests, helping identify potential performance bottlenecks or deviations from expected behavior.
- **Throughput** assesses the volume of requests or transactions processed by microservices per unit of time, indicating the system's capacity and workload handling capabilities.
- **Error Rates** tracks the occurrence of errors and exceptions, offering insights into the robustness and fault tolerance of microservices.
- **Resource Utilization** look at the use of computing resources such as CPU, memory, and network bandwidth, aiding in the identification of potential resource contention or inefficiencies
- **Latency Metrics** analyze various latency metrics, including network latency and database query latency, to pinpoint areas where delays may impact overall system responsiveness.
- **Dependency Tracking** exhibits the flow of requests by expliciting dependencies and interactions with external services.
- **Custom Business Metrics** creates instrumentation for specific business logic metrics relevant to the microservice's function.

In the CNCF landscape and common design patterns (cf. Section **??**), instrumentation metrics are usually exposed through a HTTP endpoint (commonly */metrics*). This endpoint serves as the resource from which Prometheus [18], an open-source systems monitoring and alerting toolkit, scrapes metrics. The CNCF [19] releases an open standard called OpenMetrics [20] that extends and evolves the Prometheus format. OpenMetrics specifies today's de-facto standard for transmitting cloud-native metrics at scale and brings it into an IETF standard (RFC2119 [21], RFC5234 [22], RFC8174 [23]).

3.2 Logging

Definition: *the act of keeping a record (log) of events that occur in microservices.*

In the microservices architecture, where multiple services operate independently, logs from diverse sources need to be consolidated to provide a cohesive and comprehensive view of the system's operational activities, including error messages, informational events, and performance-related details.

By aggregating logs, developers and operators gain a unified perspective on the entire microservices ecosystem, enabling efficient and effective root cause analysis in the event of issues or errors. This centralized approach enhances the overall observability, facilitating proactive monitoring and response to events that impact system functionality.

Examples of metrics obtained through log analysis and aggregation :

- **Error Rates** tracks the number of error messages or log messages with error levels (like `WARN`, `ERROR`, `FATAL`) compared to the total number of messages. High error rates can indicate problems within the system that need immediate attention.
- **Log Volume** measures the amount of log data generated over a specific period. This metric can indicate the overall activity level of the system and help in identifying unusual spikes or drops in the system activity, which could signify issues or changes to be taken care of.
- **Top N Errors** identifies the most frequent error messages or types of errors. This helps in pinpointing recurring issues or common failures in the system.
- **Latency Metrics** for systems where log generation and transmission are time-sensitive, measuring the latency between log generation, transmission, and aggregation is vital.
- **Message Sources** are metrics related to the sources of log messages, like the number of events logged by each service, application, or server. This helps in understanding which parts of the system generate most issues.

3.3 Real-Time Monitoring

Definition: *Continuously tracking and collecting data on the runtime behavior of microservices.*

Real-time monitoring involves the continuous and immediate tracking of the operational parameters and performance metrics of the system during runtime. It serves as a proactive mechanism for swiftly detecting anomalies, performance bottlenecks, and other potential issues. By capturing and analyzing data on resource utilization, response times, and other critical metrics in real-time, monitoring enables rapid identification and resolution of issues before they escalate. It empowers system administrators, developers, and automated processes to make informed decisions and take timely actions to maintain the overall health and efficiency of the microservice architecture.

As described in Section 2.3, K8S natively embeds a collection of monitoring statistics. Among the most significant are:

- **CPU and memory usage** measure the amount of CPU and memory resources consumed at the node, pod and container levels.
- **Network I/O** monitors the network bandwidth used by pod. This includes data on the amount of network traffic sent and received, which is critical for understanding the communication and bandwidth usage.
- **Network packet drops and errors** at the pod level indicate potential network issues affecting communication.
- **API Server Metrics** monitors the performance and health of the Kubernetes API server, including request rates, latencies, and error rates.
- **Scheduler Metrics** tracks the efficiency and effectiveness of the Kubernetes scheduler, including metrics related to scheduling decisions, delays, and errors.
- **Cluster Component Status** monitors the health of essential Kubernetes components like etcd, controller manager, and scheduler.

3.4 Distributed Tracing

Definition: *Capture and correlation of traces of requests as they traverse multiple microservices running on distributed nodes.*

Distributed tracing provides a holistic view of the route a request takes across various service boundaries. This involves assigning a unique identifier to each request and collecting data as it propagates through different microservices. The resulting trace offers insights into the end-to-end flow of requests, including information about latency, dependencies, and potential bottlenecks.

The significance of distributed tracing lies in its ability to connect unitary events to the higher level application logic and thus facilitate performance analysis and optimization. By visualizing the interactions between microservices, developers and system operators can identify areas of latency, pinpoint dependencies, and streamline the overall request flow. This level of visibility is crucial for maintaining and enhancing the reliability and responsiveness of microservices architectures, especially in complex, distributed environments.

In practice, distributed tracing generates a detailed timeline or "trace" of individual requests including the following metrics :

- **Request Duration** measures the total time taken for a request to traverse the microservice system.
- **Service Response Time** captures the time each microservice takes to process a request, aiding in pinpointing performance issues at the service level.
- **Span Duration** examines the time spent within each microservice (span), enabling the identification of specific components causing delays.

- **Error Rates** tracks the occurrence of errors or exceptions within traced requests, providing insights into the reliability of microservices.
- **Dependency Maps** illustrates the relationships and dependencies between microservices, aiding in understanding the overall architecture's complexity.

4 Performance Metrics Analysis for Higher-Level KPI

The authors of [24] put forward the fact that low-level metrics, such as CPU and memory usage, are the most used but do not suffice for effective auto-scaling. There is a need of a higher-level metrics reflecting the application to provide a reliable auto-scaling system. Possible approaches to the identification of relevant higher-level application indicators are numerous. In the following we discuss event-driven observability (Section 4.1), metric correlation (Section 4.2), continuous optimization (Section 4.3) and FinOps (Section 4.4).

4.1 Event-Driven Observability

Definition: *Leveraging runtime microservices' events to gain insights into their operational states.*

In the microservices paradigm, where services often operate independently and asynchronously, event-driven observability leverages the generation and propagation of events to provide a dynamic view of the system behavior. These events may include changes in service state, the detection of specific activities, or responses to external stimuli. By capturing and analyzing these events, developers and operators gain valuable insights into the real-time operational dynamics of microservices. Instead of relying solely on polling or periodic checks, this approach allows for immediate responses to changes in the environment or the internal states of microservices. It enables a more dynamic and responsive form of observability, where the system can adapt in real-time to evolving conditions, ensuring timely and informed decision-making.

This method is particularly valuable in dynamic distributed architectures within the ability to gather the following metrics:

- **Event Rates** signals the frequency at which specific events occur. Examples are the number of successful database queries per second, the count of HTTP 500 error responses or unhandled exceptions.
- **Latency Events** indicates the time taken for specific operations to complete. For instance the distribution of response times for critical API calls.
- **Resource Utilization Events** reflects the utilization of compute resources such as network bandwidth during high-loads period.
- **Dependency Events** tracks interactions and dependencies between microservices such as database connectivity, API health checks or number of requests sent or received from dependent services.
- **Custom Business Events** reflects events tailored to specific business logic or functionality (e.g., number of items added to a shopping cart in an e-commerce service, number of accounts created over a period).

4.2 Metric Correlation

Definition: *Identifying and establishing relationships between different metrics to derive meaningful insights.*

Metric correlation involves the identification and establishment of relationships between different metrics to derive meaningful insights. It provides a snapshot of the system's health and performance and seeks to uncover the interconnected nature of these metrics. By analyzing how different performance indicators coalesce and influence one another, developers and operators gain a holistic understanding of the system behavior. For instance, correlating response times with resource utilization can unveil potential bottlenecks, while examining correlations between error rates and specific operations may pinpoint areas requiring attention. This technique enables a more informed and strategic perspective to system optimization, as it considers the interdependencies that shape the overall performance landscape.

Here are some examples of metrics aggregations :

- **Latency and Error Rates.** High latency may correlate with increased error rates and indicate that longer response times contribute to errors or disruptions in service.
- **Request Rate, Throughput and Resource Utilization.** Higher request rates may correlate with increased throughput and higher CPU utilization, indicating that a workload may necessitate scaling measures for optimal performance.
- **Response Time and User Satisfaction.** Longer response times may correlate with lower user satisfaction scores highlighting the impact of performance on the end-user experience.

4.3 Continuous Optimization

Definition: *Using autonomic optimization loop to feed observed data back into the system for automated decision-making or adjustment.*

Continuous optimization emphasizes the need for an ongoing and adaptive approach to system improvement. This involves leveraging observed data, metrics, and feedback loops to make informed adjustments to the microservices architecture. Continuous optimization is not a one-time event but rather a cyclical and evolving practice aimed at maintaining and improving the overall efficiency, reliability, and scalability of the system.

The essence of continuous optimization lies in its alignment with the principles of adaptability and responsiveness. By incorporating real-time observations into decision-making processes, the microservices architecture can autonomously adapt to changing conditions. This ensures that the system is consistently tuned for optimal performance, considering factors such as resource utilization, response times, and overall user experience.

This approach involves the systematic adjustment of various parameters to ensure that the ecosystem operates at its peak. Performance indicators such as resource utilization, scalability metrics or throughput allow to identify underutilized or overburdened microservices in order to optimize resource allocation and scaling strategies. Response time coupled with latency profiles pinpoint performance bottlenecks to improve user experience and overall system responsiveness. Continuous optimization relies on the iterative analysis of these and other metrics to adapt and refine the microservices architecture, ensuring it remains responsive, reliable, and cost-effective in dynamic operational environments.

In addition, compliance metrics like SLOs and SLAs are integral to the continuous optimization process. They provide a framework for setting performance goals, prioritizing improvements, aligning service performance with business objectives, and ensuring customer satisfaction.

- **SLAs**[3] are formal agreements between service providers and their clients that define the expected level of service. They often specify the consequences of not meeting these levels. SLAs establish clear expectations for customers, which in turn drive the internal objectives and targets for optimization. They ensure that the optimizations are not just technically focused but also aligned with business objectives and customer needs.
- **SLOs**[4] are specific measurable characteristics of the SLA such as availability, throughput, frequency, response time, or quality. SLOs help in prioritizing optimization tasks based on which areas are not meeting the desired objectives. They provide continuous feedback, allowing teams to iteratively improve the service.

4.4 Financial Operations

Definition: *Aligning cloud costs with actual usage and optimizing system costs to ensure financial accountability and efficiency of the service.*

Financial Operations (FinOps in short) introduce financial accountability by integrating cost considerations into the observation and optimization processes. This entails monitoring and managing the financial aspects of workloads, including resource consumption, infrastructure expenses, and overall operational costs. FinOps principles advocate for collaboration among teams, including developers, operations, and finance, to ensure that cloud resources are utilized efficiently, and costs are optimized. By incorporating financial considerations into the observation framework, organizations can make informed decisions about resource allocation, scaling strategies, and infrastructure choices. FinOps ensures that the pursuit of performance and reliability is harmonized with fiscal responsibility, contributing to the sustainability and efficiency objectives.

Example of FinOps metrics are:

- **Resource Efficiency** assesses how well microservices utilize computational, memory, and storage resources in relation to the cost incurred.
- **Data Transfer Costs** monitor the expenses associated with data transfers between microservices, especially in distributed environments, to optimize data flow and reduce unnecessary costs.

[3] Service Level Agreements
[4] Service Level Objectives

- **Downtime Costs** quantify the financial impact of downtime or disruptions, encouraging practices that minimize service interruptions and associated costs.

4.5 Concluding Remarks

The intense development of the above mentioned activities support the fact that the efficient management of microservice architectures needs mechanisms for efficient and holistic observation, as well as for autonomous decision making. These activities need to respond to multiple objectives with multiple SLOs including end user satisfaction, performance optimization, efficient resource allocation and cost- efficient development, deployment and maintenance processes.

5 Landscape of Existing Tools

The SCALER project needs to select the most appropriate tools available for Kubernetes and the cloud native ecosystem. A lot of research points out that the Cloud Native Computing Foundation (CNCF) has a wide toolbox that could correspond to the project needs. But why is it important to use or limit the choice of CNCF tools ?

Choosing CNCF projects when working on a Kubernetes project isn't strictly mandatory, but is considered a good practice for several reasons. The CNCF is an organization that fosters the growth and development of cloud-native technology ecosystems, including Kubernetes. It ensures that the projects are designed to be interoperable and compatible within the cloud-native ecosystem. It promotes standards ensuring that the components of the Kubernetes-based system work well together. The CNCF governance model usually maintains a level of vendor neutrality, reducing the risk of *lock-in* and benefits from a large and active support community that increases the guarantee that a project is maintained over the long term. Last but not least, the CNCF is at the forefront of cloud-native technologies. By choosing CNCF projects, it's likely to be working with the latest innovations in the field.

It's important to note that while there are many advantages to using CNCF projects, this doesn't mean non-CNCF projects are inferior or incompatible with Kubernetes. Many non-CNCF projects work excellently with Kubernetes and may sometimes be a better fit depending on specific project requirements and contexts. However, in order to guarantee optimal compatibility with the Kubernetes ecosystem we decide to limit our research to the CNCF landscape.

5.1 CNCF Landscape

The CNCF proposes an interactive landscape [25] that lists all cloud native open source projects and proprietary products. Figure 2 gives a capture of its observability tools. A category is dedicated to observability and analysis where we can find proprietary products (i.e. Datadog, Dynatrace, Splunk or Graylog), open source projects (i.e. Centreon, Nagios, Zabbix or Grafana) or CNCF-hosted open source projects (i.e. Prometheus, Fluentd, Jaeger or OpenTelemetry).

The CNCF classifies products into the following six sub-categories :

- **Monitoring** is the ability to continuously track and collect data about the runtime behavior of applications and systems.
- **Logging** is the process of capturing log messages that describe failed or successful action events that occur in the system.
- **Chaos Engineering** refers to the practice of intentionally introducing faults into a system in order to test its resilience and ensure applications and engineering teams are able to withstand turbulent and unexpected events.

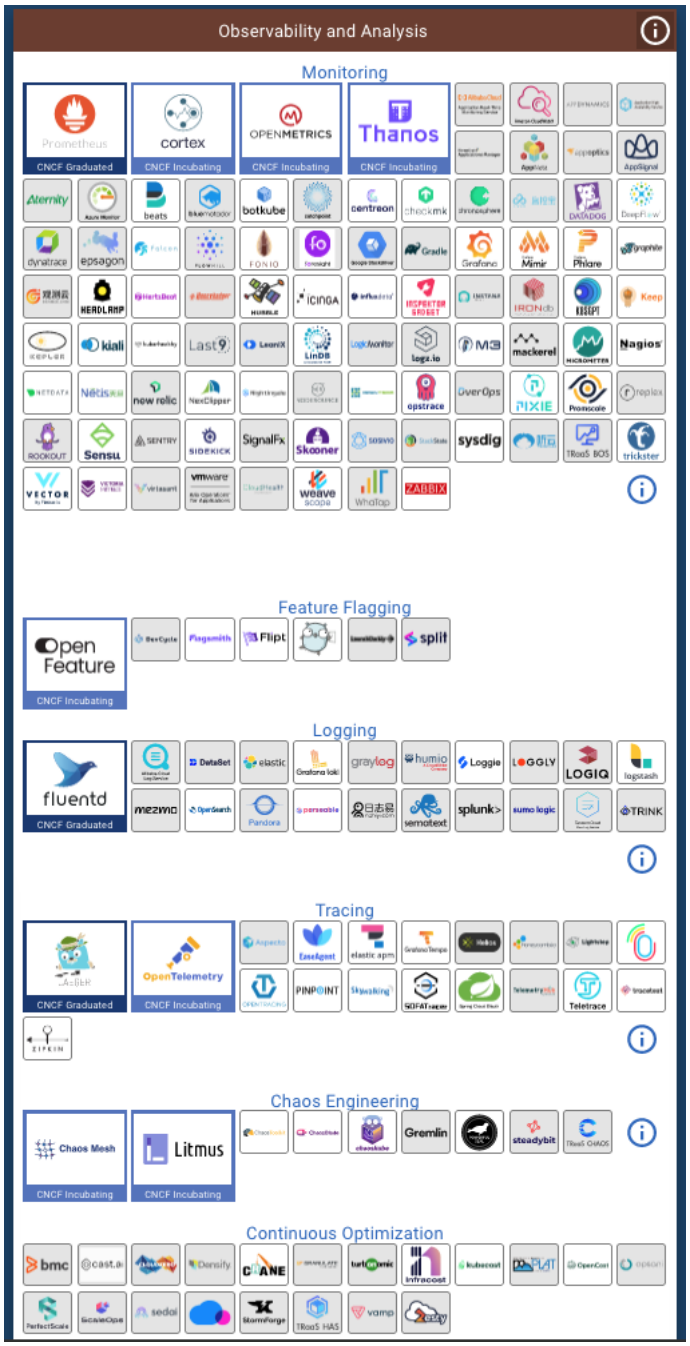

Figure 2: CNCF Observability and Analysis Landscape.

- **Tracing** allows to track the path of a request as it moves through a distributed system.
- **Continuous Optimization** establish mechanisms to feed observed data back into the system for automated decision making or adjustment.
- **Feature Flagging** lists software development tools whose purpose is to turn functionalities on or off in order to safely test in production by separating code deployment from feature release.

Two of them, *Chaos Engineering* and *Feature Flagging* are not considered as observability tools and will be put aside from our concerns. The other four categories are inventories of tools that cover

one or more of the previously disclosed approaches. Our selection is based on free open source software that is available for on-premise deployment regardless of the cloud provider. Products available as SaaS are not included for now.

5.2 Monitoring Tools

Product	µservice Specific	Open Source	Features	Usage
Centreon [26]	no	yes	monitoring, analytics	Real-time monitoring
Datadog [27]	no	no	monitoring, analytics	Real-time monitoring
Dynatrace [27]	no	no	monitoring, analytics	Real-time monitoring
Grafana [28]	no	yes	dashboard, visualization	Real-time monitoring, Instrumentation, Log Aggregation, Metric Correlation
Graphite [29]	no	yes	visualization, long term storage	Real-time monitoring, Metric Correlation
Hubble [30]	yes	yes	monitoring, analytics, security observability	Real-time monitoring, Metric Correlation, Event-Driven observability
InfluxDB [31]	no	yes	monitoring, timseseries DB	Real-time monitoring, Metric Correlation
Mimir [32]	no	yes	long term storage	Real-time monitoring
Netdata [33]	no	yes	monitoring, analytics	Real-time monitoring
Nightingale [34]	no	yes	dashboard, visualization, monitoring, timeseries DB	Real-time monitoring, Instrumentation, Log Aggregation, Metric Correlation
OpenMetrics [35]	no	yes	specifies the de-facto standard for transmitting cloud-native metrics at scale	Real-time monitoring
OpenTSDB [36]	no	yes	timseseries DB	Real-time monitoring
Prometheus [18]	no	yes	monitoring, timseseries DB	Real-time monitoring, Metric Correlation
Thanos [37]	no	yes	long term storage	Real-time monitoring
Vector [38]	no	yes	observability pipelines builder	Instrumentation, Log Aggregation, Metric Correlation
VictoriaMetrics [39]	no	yes	monitoring, timseseries DB	Real-time monitoring, Metric Correlation
VMware Aria [40]	no	no	dashboard, visualization, traces, monitoring, timeseries DB	Real-time monitoring, Instrumentation, Log Aggregation, Metric Correlation
Zabbix [41]	no	yes	monitoring, analytics	Real-time monitoring

Table 1: Monitoring Products

5.3 Logging Tools

Product	µservices Specific	Open Source	Feature	Usage
Grafana Loki [42]	no	yes	log analytics	Log Aggregation
Elasticsearch [43]	no	yes	log analytics, visualization	Instrumentation, Log Aggregation
Fluentd [44]	no	yes	log collector	Instrumentation
Graylog [45]	no	yes	log analytics, visualization	Log Aggregation
Splunk [46]	no	no	log analytics, monitoring, dashboard, visualization	Real-time monitoring, Log Aggregation, Metric Correlation

Table 2: Logging Products

5.4 Tracing Tools

Product	µservices Specific	Open Source	Feature	Usage
Grafana Tempo [47]	no	yes	trace collector, easy to use, high scale tracing backend	Distributed Tracing
Jaeger [48]	yes	yes	SPM, topology graphs, opentelemetry compatible	Distributed Tracing
OpenTelemetry [49]	yes	yes	vendor neutral, developpers toolbox (API, SDK)	Instrumentation, Distributed Tracing
skywalking [50]	yes	yes	log analytics, APM, dashboard, visualization	Real-time monitoring, Instrumentation Log Aggregation, Distributed Tracing
tracetest [51]	no	yes	easy creation of E2E tests for OpenTelemetry devlopments	Instrumentation
Zipkin [52]	no	yes	dependency diagram	Distributed Tracing

Table 3: Distributed Tracing Products

5.5 Continous Optimization and FinOps Tools

Product	µservices Specific	Open Source	Feature	Usage
Crane [53]	yes	yes	visualization, analytics, optimization	Continuous Optimization Financial Operations
Opencost [54]	yes	yes	visualization, analytics	Financial Operations

Table 4: Continous Optimization and FinOps Products

6 Concluding Remarks

This report has presented an overview of the rich landscape of observation tools applicable to the microservice context. They allow for a holistic approach to observation and provide a foundation for deriving actionable insights and metrics. Available observation data becomes the key to unlocking the understanding needed for the automatic scalability of microservices in dynamic and evolving environments.

It is important to note that most cited tools are well suited for web applications. However, most do not take into account the specific requirements of other applications, such as the 5G use case (see deliverable Scaler D4.1). New difficulties can be observed, requiring the development of new knowledge/methodologies for the near-real-time requirements of 5G and, more generally, telecom applications.

The work in this report is to serve as a foundation for the work on the analysis of the behaviour of groups of micro-services, with a view to moving towards a holistic approach over time. In addition to the application, this approach should provide a view of the underlying infrastructure [55] with possible interferences caused by the shared use of cloud infrastructures [56].

A Load Generators for Microservice Benchmarking

- Fortio (https://fortio.org/) started as, and is, Istio's load testing tool and later (2018) graduated to be its own open-source project.
- Locust (https://locust.io/) is an open source performance/load testing tool for HTTP and other protocols.

References

[1] *Kubernetes (K8s): An Open-Source System for Automating Deployment, Scaling, and Management of Containerized Applications*. https://kubernetes.io/.

[2] João Nunes, Thiago Bianchi, Anderson Iwasaki, and Elisa Nakagawa. "State of the Art on Microservices Autoscaling: An Overview". In: *Anais do XLVIII Seminário Integrado de Software e Hardware*. Evento Online: SBC, 2021, pp. 30–38. DOI: 10.5753/semish.2021.15804. URL: https://sol.sbc.org.br/index.php/semish/article/view/15804.

[3] Shamsuddeen Rabiu, Chan Huah Yong, and Sharifah Mashita Syed Mohamad. "A Cloud-based Container Microservices: A Review on Load-balancing and Auto-scaling Issues". In: *International Journal of Data Science* 3.2 (2022), pp. 80–92. DOI: 10.18517/ijods.3.2.80-92.2022.

[4] Zhiheng Zhong, Minxian Xu, Maria Alejandra Rodriguez, Chengzhong Xu, and Rajkumar Buyya. "Machine Learning-Based Orchestration of Containers: A Taxonomy and Future Directions". In: *ACM Comput. Surv.* 54.10s (Sept. 2022). ISSN: 0360-0300. DOI: 10.1145/3510415. URL: https://doi.org/10.1145/3510415.

[5] Siti Nuraishah Agos Jawaddi, Muhammad Hamizan Johari, and Azlan Ismail. "A review of microservices autoscaling with formal verification perspective". In: *Software: Practice and Experience* 52.11 (2022), pp. 2476–2495. DOI: https://doi.org/10.1002/spe.3135. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3135. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3135.

[6] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 19–33. ISBN: 9781450362405. DOI: 10.1145/3297858.3304004. URL: https://doi.org/10.1145/3297858.3304004.

[7] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. "Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 135–151. ISBN: 9781450383172. DOI: 10.1145/3445814.3446700. URL: https://doi.org/10.1145/3445814.3446700.

[8] Manuel Gotin, Felix Loesch, Robert Heinrich, and Ralf Reussner. "Investigating Performance Metrics for Scaling Microservices in CloudIoT-Environments". In: Mar. 2018. DOI: 10.1145/3184407.3184430.

[9] Nikolas Herbst, Rouven Krebs, Giorgos Oikonomou, George Kousiouris, Athanasia Evangelinou, Alexandru Iosup, and Samuel Kounev. *Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics*. Apr. 2016.

[10] Betsy Beyer, Niall Richard Murphy, David K Rensin, Kent Kawahara, and Stephen Thorne. *The site reliability workbook: practical ways to implement SRE*. 2018.

[11] Eman Aljoumah, Fajer Al-Mousawi, Imtiaz Ahmad, Maha Al-Shammri, and Zahraa Al-Jady. "SLA in cloud computing architectures: A comprehensive study". In: *Int. J. Grid Distrib. Comput* 8.5 (2015), pp. 7–32.

[12] *October FinOps Summit 2021: A 9-year story, Adopting FinOps, Reducing Waste, and Multi-Cloud updates*. https://www.youtube.com/watch?v=zFFuuZpN1Hc.

[13] *Kubernetes Tools for Monitoring Resources*. https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-usage-monitoring/.

[14] *Kubernetes Metrics Server*. https://github.com/kubernetes-sigs/metrics-server.

[15] *K8S Kubelet*. https://itknowledgeexchange.techtarget.com/coffee-talk/files/2021/03/.

[16] *Container Runtime Interface (CRI)*. https://kubernetes.io/docs/concepts/architecture/cri/.

[17] *gRPC: A high performance, open source universal RPC framework*. https://grpc.io/.

[18] *Prometheus*. https://prometheus.io/.

[19] *Cloud Native Comûting Foundation*. https://www.cncf.io/.

[20] *The OpenMetrics project*. https://openmetrics.io/.

[21] *RFC21119*. https://www.rfc-editor.org/info/rfc2119.

[22] *RFC5234*. https://www.rfc-editor.org/info/rfc5234.

[23] *RFC8174*. https://www.rfc-editor.org/info/rfc8174.

[24] Abeer Abdel Khaleq and Ilkyeun Ra. "Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications". In: *IEEE Access* 9 (2021), pp. 35464–35476. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3061890. URL: https://ieeexplore.ieee.org/document/9361549/ (visited on 10/24/2023).

[25] *CNCF Cloud Native Interactive Landscape*. https://landscape.cncf.io/.

[26] *SaaS IT Monitoring Platform for Digital Performance*. https://www.centreon.com.

[27] *Datalog*. https://www.datadoghq.com/.

[28] *Grafana*. https://grafana.com/.

[29] *Graphite*. https://grafana.com/oss/graphite/.

[30] *Hubble: Network, Service Security Observability for Kubernetes*. https://github.com/cilium/hubble.

[31] *InfluxDB*. https://www.influxdata.com/.

[32] *InfluxDB*. https://www.influxdata.com/.

[33] *Netdata*. https://www.netdata.cloud/.

[34] *Nightingale*. https://n9e.github.io/.

[35] *OpenMetrics*. https://openmetrics.io/.

[36] *OpenTSDB*. http://opentsdb.net/.

[37] *Thanos*. https://thanos.io/.

[38] *Vector*. https://vector.dev/.

[39] *VictoriaMetrics*. https://victoriametrics.com/.

[40] *VMWare Aria Network and Application Monitoring Tool*. https://www.vmware.com/products/aria-operations-for-networks.html.

[41] *Zabbix*. https://www.zabbix.com/.

[42] *Grafana Loki*. https://grafana.com/oss/loki/.

[43] *Elastic: Comprehensive and modern infrastructure monitoring*. https://www.elastic.co/observability/infrastructure-monitoring.

[44] *Fluentd*. https://www.fluentd.org/.

[45] *Graylog*. https://graylog.org.

[46] *Splunk*. https://www.splunk.com.

[47] *Grafana Tempo*. https://grafana.com/oss/tempo/.

[48] *Jaeger*. https://www.jaegertracing.io/.

[49] *OpenTelemetry*. https://opentelemetry.io/.

[50] *Apache SkyWalking*. https://skywalking.apache.org/.

[51] *tracetest*. https://tracetest.io/.

[52] *Zipkin*. https://zipkin.io/.

[53] *Crane*. https://gocrane.io/.

[54] *OpenCost*. https://www.opencost.io/.

[55] Alireza Goli, Nima Mahmoudi, Hamzeh Khazaei, and Omid Ardakanian. "A Holistic Machine Learning-based Autoscaling Approach for Microservice Applications." In: *CLOSER*. 2021, pp. 190–198.

[56] Joy Rahman and Palden Lama. "Predicting the end-to-end tail latency of containerized microservices in the cloud". In: *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2019, pp. 200–210.