Journal of Systems Architecture 145 (2023) 103035

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Monintainer: An orchestration-independent extensible container-based monitoring solution for large clusters

Miguel Correia [a], Wellington Oliveira [a,b], José Cecílio [a,*]

[a] LASIGE, Faculty of Sciences, University of Lisbon, Lisbon, Portugal
[b] COPELABS, CICANT, Lusofona University, Lisbon, Portugal

ARTICLE INFO

Keywords:
Monitoring
Container-based infrastructure
Cloud monitoring tools
Resource allocation
Internet of Things
Performance metrics

ABSTRACT

Container virtualization has recently gained popularity due to its low performance and resource allocation overhead. The rise of this technology can be attributed to the advancement of cloud computing and the adoption of micro-services architecture. These new approaches offer a more efficient and fine-grained system design through the benefits of containerization, such as isolation, portability, and improved performance. However, container-based systems have created new challenges in monitoring due to their automated flexibility, ephemerality, and the increasing number of containers in a system. So there is a practical need for effective monitoring and performance management tools. This paper analyses the key performance metrics for machine, container and application services, including CPU usage, memory usage, disk usage, and network usage. Furthermore, we review several widespread tools for collecting and monitoring these metrics and present the Monintainer tool. It is a solution designed to monitor entire container-based systems, from applications to their underlying infrastructure, allowing users to better understand their systems' behavior in run-time. The tool's results can aid container-based systems' design, implementation and optimization.

1. Introduction

In the last few years, the software virtualization paradigm has changed. Container-based virtualization technologies have seen increased adoption and growth, disrupting the typical hypervisor-based virtual machine approach. The paradigm shift is supported by containerization technology, providing isolation and greater portability. At the same time, it presents less performance and resource overhead compared to hypervisor-based virtualization, as demonstrated by Bhardwaj et al. [1] where hypervisor technology was compared to containerization technologies.

An important aspect of container-based virtualization growth is linked to the evolution of cloud computing and the search for its benefits, such as efficient allocation of resources, availability, and low infrastructure cost. Cloud providers rely on virtualization techniques to optimize resource utilization over physical infrastructure, which yields a higher return on investment. For many years, cloud solutions were predominantly based on virtual machines, but their overhead issues hinder performance and portability. Therefore, providers are focusing on lightweight techniques capable, for instance, of reducing start-up and provisioning times.

Another reason that supports containerization adoption is the system architectural shift observed in recent years. The traditional monolithic architecture that dominated software systems for decades is now being replaced by more modern architectures. For instance, micro-services, presented by Newman [2] in a holistic view, in which whole applications are decomposed into smaller and independent components, resulting in a more fine-grained system. This separation of responsibilities is potentiated through containerization characteristics such as isolation and portability, allowing micro-services to be individually managed and deployed to deal with their specific requirements, for instance, based on workloads.

Containers are highly portable components for running applications which yield the resources management and allocation flexibility that occurs in a self-managed way in run-time and in highly dynamic, distributed, heterogeneous, and unsupervised environments, as discussed by Usman et al. [3], hindering insights such as resource allocation and usage, availability, performance, and security.

The difficulty in supervising containerized systems, which is mentioned as a challenge by Casalicchio et al. [4], may incur, for instance, a low understanding of the application's behavior and containers distribution over the physical layer, broken Service Level Agreements (SLAs), degradation of the Quality of Service (QoS), higher Mean Time To Resolution (MTTR), and an inefficient energy and infrastructure usage.

The need to monitor power consumption and efficiently use energy in modern systems has become a relevant concern. As reported by

* Corresponding author.
E-mail address: jmcecilio@ciencias.ulisboa.pt (J. Cecílio).

https://doi.org/10.1016/j.sysarc.2023.103035
Received 13 February 2023; Received in revised form 3 November 2023; Accepted 24 November 2023
Available online 25 November 2023

Anders Andrea [5], it is estimated that by 2025, data centers alone could consume up to 20% of global electricity and be responsible for 5.5% of global greenhouse gas emissions. Therefore, in recent years, optimizations of resource usage have increased the efficiency of data centers and reduced their estimates of energy use [6]. However, the digital industry still has an impressive energy footprint, producing 4% of the world's greenhouse gases and increasing its energy consumption by 6% year-over-year [7].

To reduce the impact of these issues, there is a need for monitoring systems capable of supervising not only the containers and infrastructure but also the applications/services running on top of them, providing development teams and other interested parties with a holistic view of the system. This enables the capability to monitor modern systems with a greater granularity and to act over infrastructure in runtime. Other potentialities include accountability, reporting, and system maintenance.

In order to fulfill the described needs, the Monintainer monitoring approach is presented in this work. It is aimed to collect metrics from container-based systems focusing on nodes (physical or virtual), containers, and applications/services, allowing a comprehensive view of the system's behavior. Monintainer is designed to be flexible, configurable and extensible, allowing users to tailor their monitoring to their specific needs. It uses well-defined templates, providing a structured and consistent approach to define what should be monitored. The key benefit of Monintainer for container-based services includes detailed information about the system's performance at different levels through the collection of time-series data.

Monintainer provides real-time data on resource usage, which can be used to troubleshoot problems and to understand the behavior of the overall system or a specific service, to scale services (up or down), and help identify security risks and potential breaches. The metrics collected are exposed through the Monintainer API in a standard format that enables visualization and querying capabilities and can be used to act over the containerization infrastructure accordingly. For example, rearrange container usage on physical hardware or turn off hardware to optimize energy efficiency according to resource and allocation needs.

Overall, Monintainer is helpful for ensuring container-based services' reliability, performance, and security. It collects and processes the data needed to make informed decisions on optimizing the resources and troubleshooting the issues with the services. It is a transversal monitoring solution, offering support for modern systems based on technologies such as Kubernetes, Docker Swarm, Apache Mesos, and Docker, which allows for a holistic view of containerized systems, reducing the need for distinct tools to monitor multiple containerization platforms and types of metrics. Its source code is available at the authors' repository.[1]

2. Related work

There are multiple and distinct containerization platforms. The most widely adopted are Docker [8], LXC [9], and rkt [10], as stated by Bhardwaj et al. [1] where these three were compared to KVM [11] in terms of performance.

Besides the containerization platforms, there are several container-based orchestration platforms. Kubernetes [12] is the industry leader for container orchestration and is referenced across the mentioned studies as such, also characterized by Jawarneh et al. [13] as one of the most complete orchestration frameworks. Docker Swarm [14], Apache Mesos [15], and OpenShift [16] are alternatives to Kubernetes. Arguably the three most used frameworks after Kubernetes. Truyen et al. [17] stated that Kubernetes, Docker Swarm, Apache Mesos, and OpenShift are the most used platforms in OpenStack clouds, supported by a survey among Open Stack cloud administrators.

[1] https://git.lasige.di.fc.ul.pt/jcecilio/monintainer

Several tools were developed to consider the different containerization and orchestration platforms and to monitor and collect metrics from containers. Bhardwaj et al. [1] compared Docker-stats [18], cAdvisor [19], Prometheus [20], Sensu [21], and Sysdig [22]. Moradi et al. [23] evaluated ConMon, and Usman et al. [3] introduced an overview of monitoring tools in literature such as Instana [24], Dynatrace [25], Datadog [26], Prometheus, and many others. Casalicchio et al. [27] focused on the evaluation of the best tools for measuring the performance of container-based applications and the impact of Docker containers on CPU load and disk I/O throughput, where cAdvisor, mpstat, iostat, and docker-stats were used to collect performance data, Grafana [28] and Prometheus were used to extract the data from cAdvisor. The authors stated that the overhead of these tools is negligible in terms of CPU utilization. However, the lack of tools that cover a wide range of performance metrics forced the authors to use multiple tools to get a proper performance overview, which was described as a total mess.

Docker-stats is the most simple tool. It is intended to work with Docker containers and does not provide any support for visualization tools. cAdvisor is a tool created by Google that collects and exposes real-time metrics, in Prometheus format, from containers, which can later be used to visualize, for instance, in Grafana. Despite collecting I/O metrics, Casalicchio et al. [27] consider cAdvisor to be unstable for that purpose. These two target only container-wise metrics, such as CPU, memory, and network I/O usage.

Prometheus is a common and widely used open-source solution for scraping, storing, and exposing metrics in a standard format that allows for alerting, query functionalities, and visualization tools. It can also be used to monitor applications through the use of exporters, which act as an extensibility solution that enables third-party monitoring to be ingested by the Prometheus scrapping mechanism. Prometheus server is composed of the retrieval worker, an HTTP server, and a time-series database. The retrieval worker is the component responsible for pulling data from exporters or a given pushgateway and storing it in the database, which means it is the retrieval worker that must check the scrape interval for each exporter and trigger a pull event for every exporter in the system. Exporters differ from pushgateway in the sense that exporters collect data upon the pulling event triggered by the retrieval worker, instead the pushgateaway is a solution for short-lived jobs, such as batch jobs, in which jobs push the collected data to the pushgateaway, which will be used to store data before it is pulled by retrieval worker. The HTTP server is the component that communicates with PromQL clients such as Grafana and the Prometheus UI. It is responsible for checking the alerting rules and pushing events to the alert manager, which will then notify the clients. The Prometheus database is allocated locally in the system nodes, and it is not designed to scale horizontally, which becomes a major concern in growing systems. One scaling approach for Prometheus is to setup a new Prometheus instance which divides the monitoring and storing of the entire system across its instances, although this poses other limitations such as data aggregation and management overhead, which highly affects the capability for a holistic view. Furthermore, Prometheus is not a durable long-term storage solution.

ConMon is an OCI [29] run-time compatible monitoring tool to discover application containers and monitor their network performance. ConMon is also being used as a Kubernetes monitoring tool.

Sensu, Sysdig, Instana, Dynatrace, and Datadog are the tools with the most advanced features that provide support for automatic service discovery, i.e., the capability to auto-detect the applications that are running inside containers and providing appropriate metrics related to those applications, although these are commercial tools.

Table 1 presents an overview of the monitoring tools. The table shows a list of container-based orchestration platforms supported by each monitoring tool. This table shows that some solutions are targeted at specific platforms, and no single solution is available to deal with all platforms. Besides the platform support, the table includes Service-wise

Table 1
Monitoring tools overview.

	Platforms support	Service-level monitoring	Service automatic detection	Commercial product
Docker-stats	Docker	✘	✘	✘
cAdvisor	Docker	✘	✘	✘
Prometheus	Kubernetes, Docker	✔	✘	✘
ConMon	OCI run-times	✘	✘	✘
Sysdig	Kubernetes, OpenShift, Docker	✔	✔	✔
Instana	Kubernetes, OpenShift, Docker, CRI-O, containerd, LXC	✔	✔	✔
Datadog	Kubernetes, OpenShift, Docker, CRI-O, containerd	✔	✔	✔
Dynatrace	Kubernetes, OpenShift, Docker, CRI-O, containerd	✔	✔	✔
Sensu	Kubernetes, OpenShift, Docker	✔	✔	✔

monitoring and Automatic service discovery features and a classification if the tool was developed for commercial purposes. From these features, we can conclude that commercial tools are more promising. However, they are not available for testing, and we rely only on the documentation to make this comparison.

Besides those tools, other works aim to profile container-based applications. Enes et al. [30] presented BDWatchdog, a monitoring and profiling tool aimed at measuring the performance of Big Data workloads by inspecting the application level instead of the infrastructure as a whole, as it would be a black-box approach. BDWatchdog enables, through flame graphs and D3 [31] library graphs, the potential to improve code efficiency by visualizing the representations of application bottlenecks and performance patterns. The solution presented for BDWatchdog contemplates the use of a real-time monitoring pipeline composed of three layers. The bottom layer is responsible for extracting metrics through agents installed in system instances, the middle layer is where a distributed database approach is implemented, and the top layer is used for visualization through the browser, CLI, Grafana [28], and others. It is targeted to Big Data monitoring and does not provide any insight for other container-based applications, nor can it be used to understand infrastructure power efficiency.

Al-Dhuraibi et al. [32] proposed ElasticDocker, a live migration and vertical container scaling system. ElasticDocker scales container CPU and memory based on workloads and can migrate containers whenever the host machine cannot cope with container resource requirements. For this purpose, the resource limitation is done through Linux control groups (cgroups), and the live migration is achieved through the Checkpoint/Restore in Userspace (CRIU) tool. The tests performed by the authors show a considerable improvement in execution and provisioning times compared to Kubernetes horizontal auto-scaling and a little migration time. The authors conclude that the improvements introduced by ElasticDocker translate into a better Quality of Experience (QoE) and lower provisioning costs. An improvement to this work is the inclusion of intelligent systems for planning, scaling workloads, and container migration. The ability to monitor applications may also enable more fine-grained monitoring consumed by planning and scaling mechanisms.

Casalicchio et al. [33] introduced a study on the impact of relative and absolute metrics for auto-scaling containers. Absolute metrics cover the resource utilization of the host system, instead relative metrics are container-wise. Known platforms such as Kubernetes rely on relative metrics to implement auto-scaling. However, the authors consider absolute metrics to be critical for improving aspects, such as response times and meeting SLA requirements. KHPA-A is an auto-scaling system based on fundamental metrics presented by the authors. It is intended to be plugged into Kubernetes and accepts parameters for utilization thresholds. The study demonstrates that auto-scaling based on absolute metrics substantially improved the response times of applications. However, this work focuses only on CPU utilization and the impact of metrics in Kubernetes. A broader approach is still missing regarding other types of workload and orchestration frameworks.

Grossmann et al. [34] introduced PyMon, a monitoring tool designed for container-based computing architectures with a small resource footprint. The focus of the tool is on monitoring Edge nodes, with each node hosting a monitoring probe that sends data to a centralized time-series database. Another Edge monitoring tool based on container solutions is FMonE, proposed by Brand'on et al. [35]. FMonE is designed to achieve resiliency and elasticity for Edge systems. Similarly, works such as [36,37] are tagged to be lightweight, offering a limited set of metrics and lacking mechanisms for extension.

Measuring the energy consumption of containerized ecosystems has been the focus of other monitoring tools. For instance, Piraghaj et al. [38] proposed a formula for estimating power consumption in cloud data-center consolidation based on cluster CPU usage. However, this solution presents a simple approach that does not take into consideration factors such as I/O operations and memory and does not provide the means to isolate the consumption of an individual container within the cluster.

Another commonly applied solution is collecting power consumption using an external specialized hardware device. Many devices take these measurements, some of which have been used to analyze the consumption of servers/data centers (Yocto-Watt [39] by Kang et al. [40]; RCE PM600 [41] by Tadesse et al. [42]; HP Intelligent Modular PDU [43] by Tesfatsion et al. [44], and Watts up? Pro by Santos et al. [45], and Jiang et al. [46]). These devices accurately report power consumption at a given time. However, these tools' measurements encompass the power consumption for the whole machine, and it is non-trivial to isolate the energy consumption containers or applications.

Along with these tools, there are other strategies based on energy consumption estimation using internal hardware components, such as Intel Running Average Power Limit (RAPL) [47]. RAPL consists of low-level interfaces that provide information about energy and power consumption using a software power model. RAPL calculates power consumption using hardware performance counters and input and output models.

RAPL has also been used to collect energy consumption of containerized environments [48–50]. Monintainer will measure and profile

the energy consumption using the RAPL tool for the machine and the container levels, and it uses the formula defined by Piraghaj et al. [38] for the application level.

The mentioned works [30,32,33,51] limit the scope of monitoring to types of workloads, types of resources, or components, and lack some form of extensibility. Our proposed system aims to overcome this limitation in monitoring scope by monitoring the entire system and, by default enabling support for multiple types of applications. In addition, it is intended to support a broader scope of metrics, not limiting the monitoring to CPU or memory resources, and to provide an interface for custom metrics configuration. Moreover, our proposed system offers the capability to monitor and profile energy consumption.

Monintainer differs from the current state-of-the-art because it does not rely on pull-based approaches since the monitor and agents report their data to upper components. Monintainer is a fully modular approach which allows each component to be decoupled, managed, and allocated independently. This approach potentiates the components' horizontal scaling without the overhead of managing schedulers and states among replicas. Additionally, the scraping and reporting parts were designed to provide flexibility to deal with different use cases.

3. Metrics

In order to achieve a better understanding of the common metrics for container-based systems, research was conducted on the metrics supported by the aforementioned monitoring tools. Based on that research, an overview of the most valuable metrics was formulated. The purpose of the described metrics is to cover nodes (physical or virtual), containers, and applications, which enables a more insightful and fine-grained monitoring solution.

Infrastructure-related metrics:

- Node (physical or virtual)
  - Logical CPU cores - Count of logical CPU cores
  - Physical CPU cores - Count of physical CPU cores
  - CPU usage - Current amount of CPU consumed (%)
  - CPU power consumption - Current amount of power that CPU is consuming (Watts)
  - CPU energy consumption - Current amount of energy consumed by CPU (Joules)
  - CPU temperature - Current CPU temperature (Celsius)
  - GPU usage - Current amount of GPU consumed (%)
  - GPU power consumption - Current amount of power that GPU is consuming (Watts)
  - GPU energy consumption - Current amount of energy consumed by GPU (Joules)
  - GPU temperature - Current GPU temperature (Celsius)
  - Memory usage - Current amount of memory consumed (%)
  - Disk usage - Current amount of disk consumed (%)
  - Up-time - Current amount of time elapsed since the start-up (seconds)
  - Containers - Count of containers running
  - Ports - Count of open ports
  - Cluster size - Current size of the node's cluster
- Containers
  - CPU usage - Current amount of CPU consumed (%)
  - CPU energy consumption - Current amount of energy consumed by container in CPU (Joules)
  - Memory usage - Current amount of memory consumed (%)
  - Disk usage - Current amount of disk consumed (%)
  - Disk limit - Maximum amount of disk allocated to the container (bytes)
  - Memory limit - Maximum amount of memory allocated to the container (bytes)
  - Memory exceeded - Count of times that memory exceeds the defined memory limit since container start-up
  - Packets received - Count of packets received since container start-up
  - Packets dropped - Count of packets dropped since container start-up
  - Up-time - Current amount of time elapsed since the container start-up (seconds)
  - Threads - Count of threads running inside the container
  - Processes - Count of processes running inside the container
  - Threads limit - Maximum number of threads allowed to run inside the container
  - Ports - Count of container exposed ports

Application-related metrics:

- Web servers
  - Connections - Count of active client connections waiting for a response
  - Requests - Count of requests in the last second
  - Successful requests - Count of successful requests (2xx codes) in the last second
  - Client errors - Count of errors (4xx codes) in the last second
  - Server errors - Count of errors (5xx codes) in the last second
  - Throughput - Average amount of kBytes (response payload) served in the last second
  - Up-time - Current amount of time that service was available (seconds)
  - Response time - Average response time for requests in the last second (seconds)
  - Peak response time - Highest response time for request in the last second (seconds)
- Databases
  - Connections - Count of open connections
  - Most connections - Highest number of connections at a given moment
  - Connections limit - Maximum connections that can be handled
  - Refused connections - Count of connections refused due to the connections limit in the last minute
  - Selects - Count of select queries in the last minute
  - Updates - Count of update queries in the last minute
  - Inserts - Count of insert queries in the last minute
  - Deletes - Count of delete queries in the last minute
  - Errors - Count of query errors in the last minute
  - Reads - Count of read requests in the last minute
  - Writes - Count of write requests in the last minute
  - Up-time - Current amount of time that service was available (seconds)
  - Response time - The average response time for transactions in the last minute (seconds)
  - Peak response time - Highest response time for transactions in the last minute (seconds)
- IoT Brokers
  - Connections - Count of active connections
  - Messages - Count of active messages in the broker
  - Topics - Count of active topics
  - Subscriptions - Count of active subscriptions
  - Producer events - Count of published messages in the last second
  - Subscriber events - Count of pulled/pushed messages in the last second
  - Up-time - Current amount of time that service was available (seconds)

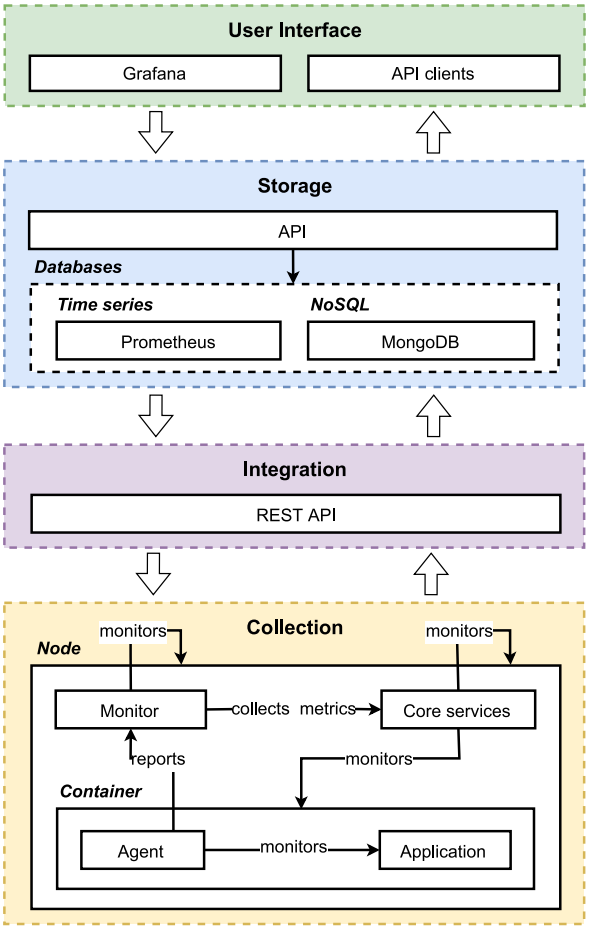


Fig. 1. Monintainer architecture.

4. Monintainer approach

Monintainer is an approach for monitoring container-based systems. It is designed to allow for a holistic view of the system, which can be used to perform a more informed correlation of the components of the system to better describe its behavior and usage.

A container-based system is typically formed by multiple micro-services. It is managed by orchestration frameworks, which distribute the system's load across a set of nodes allocated to a given cluster through the deployment of new containers. Given the components that incorporate this kind of system, Monintainer aims to monitor the system infrastructure, such as cluster nodes, deployed containers, and the services exposed by the system. This approach enables a whole-system monitoring view with greater granularity and improves the system profiling by allowing a broader monitoring scope.

Monintainer supports monitoring applications/services such as web servers, databases, and brokers out of the box. Although container-based systems are known to be heterogeneous and highly dynamic. For that reason, there is a need for custom metrics support. This support provides a way to expand monitoring capabilities, such as support for new metrics that can be used to monitor different applications, posing a more robust and interoperable monitoring system.

An insightful monitoring system requires visualization and analysis techniques, which can be achieved using multiple third-party tools. To support integration with external tools, Monintainer focuses on exposing standard formats for metrics, such as the Prometheus format. This enables users to create their dashboards and perform a specific data correlation rather than presenting a fixed data visualization solution that would not be suited for all user requirements.

Monintainer architecture is made up of four layers responsible for data collection, integration, storage, and visualization. Each layer has its specific purpose designed to fulfill the described requirements and to improve scalability. Any sub-component of Monintainer can be scaled horizontally using load balancers, distributed databases, or stateless agents that send their data to the storage component, which allows users to monitor large clusters without complex data mining systems, where data must be correlated and analyzed. The architecture of the Monintainer components is illustrated in Fig. 1.

The collection layer comprises the node monitor, the core services, and the container agent that are deployed on each node (physical or virtual) to collect the system metrics. Each component is used to monitor different levels of service in the system. However, the node monitor aggregates the collected data, which may later be used to correlate monitors and their respective reporting agents.

Besides data aggregation, the node monitor collects node metrics using programs such as psutil, fetching core services data and communicating with the integration layer to persist collected data and fetch its configurations.

Core services consist of side tools dedicated to collecting specific metrics, such as container-related metrics and energy usage, for example, through cAdvisor and Powerjoular [52] instances. Core services expose their metrics and are periodically fetched by the node monitor. Since no single tool allows collecting all the metrics at once, Monintainer can act as an aggregator at the agent level by aggregating metric data from other tools. However, this aggregation is done according to the metric configuration, providing a unified approach seamlessly integrating other tools. Additionally, the Monintainer agent offers a unified interface for installing, configuring, and operating those external tools, allowing easy access and utilization without worrying about their installation and configuration complexities. Monintainer hides these details, providing a streamlined and user-friendly experience that enables users to focus on the metrics data rather than the tools used to collect it.

The container agent is responsible for fetching its configuration in run-time, collecting application-related metrics, and periodically formatting and forwarding its collected data to the monitor. It may be a wrapper for the application's logs or a more complex program that intercepts the application traffic/data and collects metrics related to that traffic or data. Monintainer assumes that each agent only monitors one application. So, if a container runs more than one different application, an agent must be deployed per application inside the container. All node, container, and application metrics supported by Monintainer were described in Section 3.

The integration layer comprises a REST API that enables access and management of resources such as agent configurations, metrics, and remote repository configurations that can be used to fetch configurations from external sources. The API defined comprises three main services: the repository, agent, and metric management services. These services provide the necessary tools for defining new configurations for both monitor and agent components and updating or removing existing configurations from these components. By basing these configurations on the templates defined for the collection layer, we ensure consistency and compatibility across the system.

In addition to these core services, we have also designed specific services exclusively focused on data. These services will look at the metric set provided by core services, filter the metrics by name and generate a lightweight JavaScript Object Notation (JSON) output. Then, the monitoring data generated is sent through the REST API to the integration layer.

This API serves as the interface between the monitoring and persistence layers, allowing us to easily send data to a single reception point without worrying about how the data will be processed later.

This approach also enables us to scale the number of agents feeding data transparently as long as the persistence layer properly ingests the data. Furthermore, because HTTP protocols are used with REST APIs to send the data, lossless stream compression can be used to reduce the size of the stream that is sent through the network, provided that the persistence layer supports it.

Once the JSON documents reach the persistence layer, they are interpreted and stored correctly as time series points, which ensures the data is accurately and efficiently stored, ready for further analysis and processing.

Monintainer outlines two configuration schemas, the monitor/agent configuration and the metric configuration. The monitor/agent configuration establishes the configuration for each monitor or agent (Listing 1). It provides a generic template that can be applied to configure various types of data collection.

```
{
    “name": “agent name",
    “target": “node|container|application",
    “status": “active | paused | stopped",
    “execution": {
        “type": “once | periodic | continue",
        “startjob": “starting_instant",
        “duration": “interval",
    },
    “endpoint": {
        “hostname": “endpoint_hostname",
        “ip_address": “ip address
                        of the endpoint",
        “port": “TCP port to connect
                    to the endpoint",
    },
    “metrics": [list of metrics
                to be collected],
    “merge": “true | false",
    “options": “other_option",
}
```

Listing 1: Template for monitor/agent configuration

The monitor/agent configuration comprises a name, a monitoring target that can be one of three types: node, container, or application, and a status that controls the execution mode: active, paused, or stopped.

The execution aspect of the monitor defines three essential details:

- The recurrence type determines if the agent will run once, continuously, or periodically (*type*).
- The start time for monitoring (*startjob*).
- The duration of the agent's operation for both the once and periodic types (*duration*).

Once the data is collected, the agent can change its own status. For periodic agents, they will be paused after metric collection, while for once agents, they will be stopped. The endpoint to report the collected data and the list of metrics to be collected are also specified in the configuration. Then, a merge field specifies if the monitor or agent should wait for all metric evaluations before reporting to the target endpoint or report as soon as a single metric is collected. Lastly, the configuration also includes an optional field for added flexibility and future extensions to Monintainer.

The metric configuration is outlined in Listing 2. Each metric definition includes a name, the frequency at which the monitor or agent should collect the metric, the buffer size which determines the number of samples to be collected before reporting, and the tool to be used for scraping the metric. The tool field specifies the tool name, its arguments, and a structured alias for the metric. It could be Monintainer, which provides support for its metrics, or another external tool, such as Prometheus exporters or Datadog. There is also an optional field called labels, allowing customized labeling to filter data during consolidation at the storage layer.

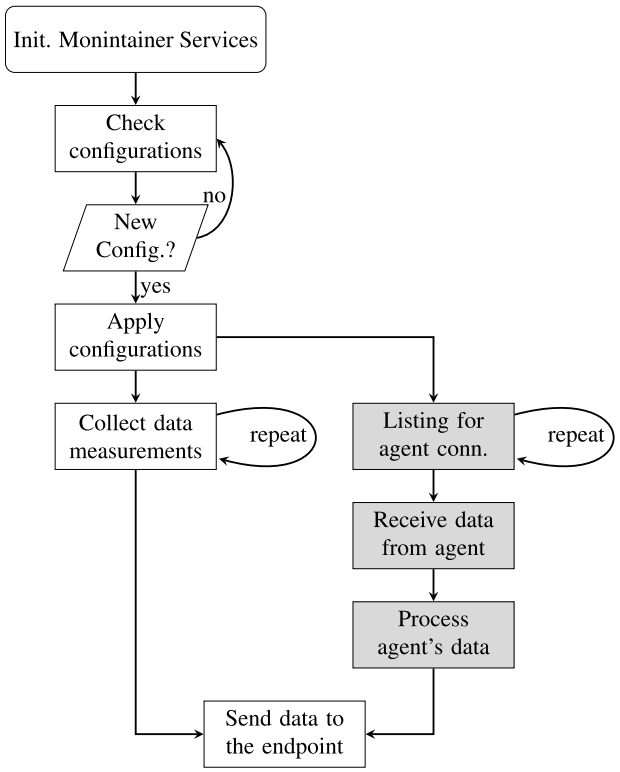


Fig. 2. Operation diagram (gray blocks are not applied to the Agent).

```
{
    “metric": “Monintainer metric name",
    “periodicity": “interval between samples
                    in milliseconds",
    “buffer": “number of samples to
                    collect before reporting",
    “tool": {
        “name": “external tool name used
                    to collect metric data",
        “args": [list of args to be passed
                    to the external tool],
        “metric_name":
            “tool_metric_name",
    },
    “labels": [list of labels to
                support data filtering],
}
```

Listing 2: Template for metric definition

This template provides flexibility to collect measurements from different tools in different ways. For example, to deal with a specific tool exposing a metric through a REST API, Monintainer must be configured with the tool name (e.g., Prometheus Node exporter) and the endpoint where the data is available is passed on the args field (e.g., *–endpoint https://<Node-exporter-IP>: <Node-exporter-PORT>/metrics*).

Fig. 2 presents the architecture of the monitor and the agent. Both processes follow the same flowchart: doing the initial setup, loading the configurations, collecting data, and sending the data to an endpoint.

The initial setup consists of initializing the core services, such as cAdvisor and the internal instruments of Monintainer, and starting a new thread that will periodically check if a new configuration is available in the repository presented in the storage layer. All initial arrangements are loaded at this point. Then, an interpreter looks at the configurations, applies them, and collects the metrics according to the specification. The collection procedure is done periodically according

to the metric specification. Upon each data collection, the data is sent to the endpoint specified in the initial configuration (using the template presented in Listing 1).

In addition to these functions, the monitor component implements another thread that listens for agent connections. When an agent sends the data, it must receive and process that data to fit the data structure defined by the integration layer. These additional parts are represented in gray in Fig. 2.

Listing 3 shows the metric template used to pass data to the integration layer. This template includes the name of the metric (the same name used to define the metric when filled in the template of Listing 2), a timestamp that represents the measurement acquisition instant, the value of the corresponding metric, and the information related with the labels specified for the metric. For instance, the labels can comprise information about the host machine, IP address or current user.

```
{
    “metric": “Monintainer metric name",
    “timestamp": “local timestamp",
    “value":metric value,
    “lables": {
        “lable_1": “lable_1_value",
        “lable_2": “lable_2_value",
    },
}
```

Listing 3: Definition of a metric value

The storage layer takes care of storing and granting access to all data. It is composed of a Prometheus [20] database, which is used to store metrics, and a MongoDB database, which stores data such as agent configuration and processed metrics. This solution allows us to use Prometheus to store and expose time-series data, which enables functionalities such as filtering and aggregation, allowing, for instance, to formulate higher-level queries, such as getting the average CPU consumed by a set of containers during a specific time window or determining the average aggregate disk write bandwidth for a container.

The combination of a MongoDB database with the Prometheus tool is used to properly manage the sparse nature of time-series points and the number of values and tags generated over time. In terms of the storage needed, it is directly proportional to the number of nodes, containers, and applications in the cluster. So, it depends on the amount of data generated. Both components are configured to be distributed, fault-tolerant, and easily scalable databases. The MongoDB database used by Monintainer is configured for data sharding to achieve these properties, which allows it to grow efficiently by adding more nodes and storage resources.

The user interface layer retrieves data and can be used to perform application profiling in the form of graphical representations. Monintainer uses the Grafana tool for that purpose, which allows you to query, visualize, and correlate metrics. Although this component can be extended through the use of API integrations, which enables clients to use alternative visualization tools and programmatic solutions that take advantage of the system monitoring.

Monintainer, contrary to other works, does not rely on approaches based on pulling, it is the agents that take the responsibility of reporting their data to upper components, in the case of the node monitor, directly to the integration layer, and for the container agent, through the report to node monitor. Both report by pushing data to their target endpoints, avoiding the need for a centralized component responsible for maintaining a global scheduler to pull data from what is likely to be an immense number of data-collecting sources with multiple granularities. It also potentiates the components' horizontal scalling without the overhead of managing schedulers and state among replicas.

Complementary, the collection components can autonomously collect and buffer data before forwarding it to the upper components, which can be used to reduce network traffic in large-scale systems without affecting granularity. Also, collection agents are automatically updated after changes applied to agent configurations, which reduces the need to directly access the infrastructure at every change. Moreover, it can be used to automatically manage agents' lifecycle, which is also a key difference from other works, avoiding the need for manually updating agents, although it does not cover edge cases where direct access to infrastructure is mandatory for dependencies installation.

In summary, Monintainer components report their data to upper components in a push-based way. Each monitor reports data directly to the integration layer, while each agent sends it to an associated monitor. Both report by pushing data to their target endpoints, avoiding the need for a centralized component responsible for maintaining a global scheduler to pull data from what is likely to be a high number of data-collecting sources with multiple granularities.

Contrary to the Prometheus tool, Monintainer is fully modular, which allows each component to be decoupled, managed, and allocated independently. Another key aspect of Monintainer is that scraping and reporting are configurable, which allows users to set up the appropriate structure for each use case. For instance, to enable the possibility of performing consolidation of the collected data transparently, mitigating the difficulty of querying data when data is generated from multiple nodes.

Both monitor and agents can scrape data from built-in metrics, command-line commands, and Prometheus-compliant exporters, providing the integration/use of many possible monitoring solutions. They can collect and buffer data autonomously before being sent to the upper components, which can be used to reduce network traffic in large-scale systems without affecting granularity. They are automatically updated after changes are applied to their configurations, reducing the need to access the infrastructure at every change. Additionally, it can be used to automatically manage monitors, agents, and exporters' lifecycles, which is also a key difference from Prometheus, avoiding the need for manual updates.

5. Monintainer evaluation

This work presents the Monintainer, a profiling solution that comprises a set of tools to perform an in-depth analysis of containerization applications/services. The first result we collected is related to comparing the most advanced monitoring tools with the proposed solution. That comparison is presented in Table 2. From the analysis of Table 2, we can conclude that Monintainer advances the state-of-the-art by providing a complete solution to evaluate the performance of a container-based solution, independently of the technology behind the container and the orchestration, as well as the application/service that is running in the container.

Considering the same monitoring tools and their metrics, Tables 3 and 4 present the comparison between them and the Monintainer approach. Note that we dropped the Sensu tool [21] from this analysis. Since it is a commercial tool without detailed documentation, it is not easy to understand what metrics and applications are supported and at which level they are available.

Following the definition of the Monintainer components, a prototype implementation of the monitor and its agent was implemented. To test the degree of technological support of Monintainer, a set of Docker containers [8] was created through Kubernetes [12], Docker Swarm [14], and Marathon [53] on top of Apache Mesos [15]. This setup allows us to evaluate the monitoring capabilities and demonstrate the support of multiple containerization systems, showing the ability of Monintainer to be used with different container technologies.

Table 2
Features comparison.

	Monintainer	Sysdig	Instana	Datadog	Dynatrace	Sensu
Service monitoring	Yes	Yes	Yes	Yes	Yes	Yes
Metrics extensibility	Yes	N/S	N/S	Yes	N/S	Yes
Product	Open-source project	Commercial	Commercial	Commercial	Commercial	Commercial
Platforms support	Kubernetes, Apache Mesos, Docker Swarm, CRI-O, Docker, containerd	Kubernetes, OpenShift, Docker	Kubernetes, OpenShift, Docker, CRI-O, containerd, LXC	Kubernetes, OpenShift, Docker, CRI-O, containerd	Kubernetes, OpenShift, Docker, CRI-O, containerd	Kubernetes, OpenShift, Docker

Table 3
Node and container metrics comparison.

		Monintainer	Sysdig	Instana	Datadog	Dynatrace
Node	Logical CPU cores	✔	✔	✘	✔	✔
	Physical CPU cores	✔	✘	✘	✘	✘
	CPU usage	✔	✔	✔	✔	✔
	CPU power consumption	✔	✘	✘	✘	✘
	CPU energy consumption	✔	✘	✘	✘	✘
	CPU temperature	✔	✘	✘	✘	✘
	GPU usage	✔	✘	✘	✘	✘
	GPU power consumption	✔	✘	✘	✘	✘
	GPU energy consumption	✔	✘	✘	✘	✘
	GPU temperature	✔	✘	✘	✘	✘
	Memory usage	✔	✔	✔	✔	✔
	Disk usage	✔	✔	✘	✔	✔
	Up-time	✔	✔	✘	✘	✘
	Containers	✔	✔	✔	✔	✔
	Ports	✔	✘	✘	✘	✘
	Cluster size	✔	✔	✔	✔	✔
Container	CPU usage	✔	✔	✔	✔	✔
	CPU energy consumption	✔	✘	✘	✘	✘
	Memory usage	✔	✔	✔	✔	✔
	Disk usage	✔	✔	✘	✔	✘
	Disk limit	✔	✔	✘	✔	✘
	Memory limit	✔	✔	✘	✔	✘
	Memory exceeded	✔	✘	✘	✘	✘
	Packets received	✔	✘	✔	✘	✘
	Packets dropped	✔	✘	✔	✘	✘
	Up-time	✔	✘	✘	✔	✘
	Threads	✔	✔	✘	✔	✘
	Processes	✔	✔	✘	✘	✘
	Threads limit	✔	✘	✘	✔	✘
	Ports	✔	✘	✘	✘	✘

5.1. Experimental setup

The underlying infrastructure to test the implementation was established using a cluster that comprises multiple machines. Each machine represents a node in a cluster with 4 GB of RAM, 2 CPU cores, and a disk size of 25 GB. The machines were set up with Ubuntu 22.04 for the Kubernetes and Docker Swarm nodes and Ubuntu 16.04 for the Apache Mesos nodes. The distinction in the Ubuntu version is due to the limitations of Apache Mesos and Marathon, which were incompatible with Ubuntu 22.04. Besides the cluster, another machine was used to act as a client supporting the execution of specific benchmarks to be used to introduce load in the cluster to collect performance data using the Monintainer approach.

Each orchestration framework was configured with a container/pod for each deployed service. The deployments were configured to run single replicas of three services: MySQL, Nginx, and Mosquitto.

Each cluster was packed with the monitor and container agent of Monintainer, and two core services, cAdvisor and Powerjoular, running in background mode through the systemd. The Monintainer monitor and container agent were implemented, according to the architecture represented in Fig. 2, as Python scripts and run as normal processes in the machine or inside the container. These Python scripts implement the engine needed to interpret the configuration files and collect metrics by executing/calling different tools and libraries, scrapping cAdvisor, and Powerjoular collected metrics.

For application-related monitoring, a MySQL service was used. The Monintainer's agent collects all metrics presented in Section 3 related to a database application. It is responsible for periodically querying the MySQL schema tables to get performance metrics and send them to the monitor.

5.2. Results

The development and setup of the aforementioned components faced some limitations. The nodes running Apache Mesos and Marathon faced limitations due to the lack of support for cAdvisor in Ubuntu 16.04. This limitation invalidates using cAdvisor for Apache Mesos nodes, which gathers container-related metrics.

Another identified barrier, which impacts the representation of the results, was the fact that cAdvisor does not export CPU, memory, and disk usage in relative percentage values. They are available in absolute

Table 4
Application metrics comparison.

		Monintainer	Sysdig	Instana	Datadog	Dynatrace
Web Server	Connections	✔	✔	✔	✔	✔
	Requests	✔	✔	✔	✔	✔
	Successful requests	✔	✘	✔	✔	✔
	Client errors	✔	✘	✔	✔	✔
	Server errors	✔	✔	✔	✔	✔
	Throughput	✔	✔	✔	✔	✔
	Up-time	✔	✘	✔	✔	✔
	Response time	✔	✔	✔	✔	✔
	Peak response time	✔	✘	✘	✘	✘
Databases	Connections	✔	✔	✔	✔	✔
	Most connections	✔	✘	✔	✔	✘
	Connections limit	✔	✔	✘	✔	✘
	Refused connections	✔	✘	✔	✘	✔
	Selects	✔	✘	✔	✔	✔
	Updates	✔	✘	✔	✔	✔
	Inserts	✔	✘	✔	✔	✔
	Deletes	✔	✘	✔	✔	✔
	Errors	✔	✘	✔	✔	✘
	Reads	✔	✘	✔	✔	✔
	Writes	✔	✘	✔	✔	✔
	Up-time	✔	✔	✘	✘	✘
	Response time	✔	✘	✔	✔	✘
	Peak response time	✔	✘	✘	✘	✘
IoT Brokers	Connections	✔	✘	✘	✔	✘
	Messages	✔	✘	✔	✔	✔
	Topics	✔	✘	✘	✘	✘
	Subscriptions	✔	✘	✘	✘	✘
	Producer events	✔	✘	✔	✔	✔
	Subscriber events	✔	✘	✔	✔	✘
	Up-time	✔	✘	✘	✘	✘

values, such as seconds and bytes. This hinders the representation of results for container-related metrics and imposes the need for post-gather processing.

Overcoming these limitations, the setup was built. The tool used for MySQL benchmarking was sysbench [54]. It is a complete benchmark that allows understanding the behavior of the MySQL components in more realistic experimentation. So, this work was associated with Monintainer, enabling us to evaluate whether the metrics collected through Monintainer are correct.

Sysbench was executed on the client node. It was configured to run four threads, which generated the service load within a defined period of 3 min, starting 15 s after the Monintainer monitor and agent start and finishing about 45 s before the end of the period, which translates to execution of about 2 min. This process was repeated five times for each platform separately.

The monitor and the agent collected data from nodes and services while the benchmark was running. The instant in which data is collected is called a tick, and the interval between ticks is the granularity of data collection. In this case, since the monitored service is a MySQL database, where most of the metrics are cumulative values, a lower granularity was assumed and the tick was defined as 1 s.

Table 5 summarizes the metrics currently collected by Monintainer to profile the MySQL performance application. These results are similar to the ones collected by Maqbool, J. et al. in [55], which indicates that the results are accurate. At the same time, we can conclude that the presence of Monintainer in the machines does have a negligible overhead over the application's performance.

A subset of collected metrics was then used to produce the graphs presented in Figs. 3 and 4 for the nodes and Fig. 5 for the application. These graphs are examples of representations that can be achieved through the data collection performed. In this case, they are related to resource usage and application throughput. Therefore, their purpose is to demonstrate the representation capabilities of Monintainer.

Our experiment can be divided into three different phases: (i) **idle**, (ii) **preparation**, and (iii) **execution**. The aforementioned graphs present each of these steps and show how Monintainer can detect subtle variations in the execution.

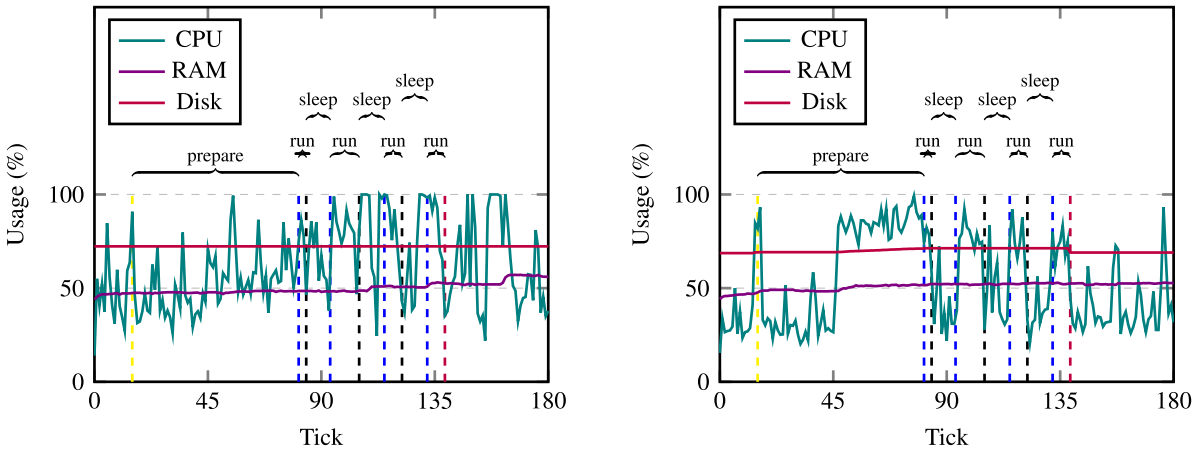
The first phase, **idle**, starts before the application begins. This phase aims to start the measurements and reduce the probability of collecting incorrect data. Although there is no application running, Monintainer starts collecting the available metrics.

The **preparation** starts when the application execution begins at 15 ticks. This is marked in the graphs with a yellow dotted vertical bar. After starting, there is a period called (*prepare* in the graphs), in which the application is getting ready to run. In the first half of this period, no operations were being executed by the application, which is reflected on the graphs with a lower CPU usage. In the second half, once the MySQL application starts adding elements to the database, there is a surge in CPU usage. This effect is particularly noticeable on the worker node. This phase takes approximately 60 ticks.

In the third phase, **execution**, invocations to sysbench are made and then the application rests for 10 ticks. The run periods are marked in the graphs with a blue dotted vertical bar (*run* in the graphs). Similarly, the sleep periods are marked in the graphs with a black dotted vertical bar (*sleep* in the graphs). Each run period executes a different number of invocations to sysbench: the first run period uses sysbench a single time; the second, three times; the third and fourth, two times. Each invocation accounts for 100 events.

As the operations are executed during the run periods, application collected metrics related to the number of operations (i.e., the number of executed Selects, Inserts, Updates, and Deletes on the database) only increases during run periods, as seen in Fig. 5. The effect of invocation sysbench once, twice, or thrice can be seen on the slope of each line. Likewise, during rest periods, there is no slope. The operations execution then causes an intense CPU usage on the run periods, contrasting with the low usage of the sleep periods, as seen in Figs. 3 and 4. This phase takes approximately 55 ticks.

The last phase is also **idle** and starts at approximately 140 ticks. This is marked as a red dotted vertical bar on the graphs. All operations in the database are halted and the cleanup process starts. Notwithstanding, there are still running processes on the node, which are detected by Monintainer. An example of such a process is Kubernetes orchestration. Even after the cleanup, the master node is still working on orchestrating



(a) Master node

(b) Worker node

Fig. 3. System usage for Kubernetes.

Table 5
Summary of collected metrics.

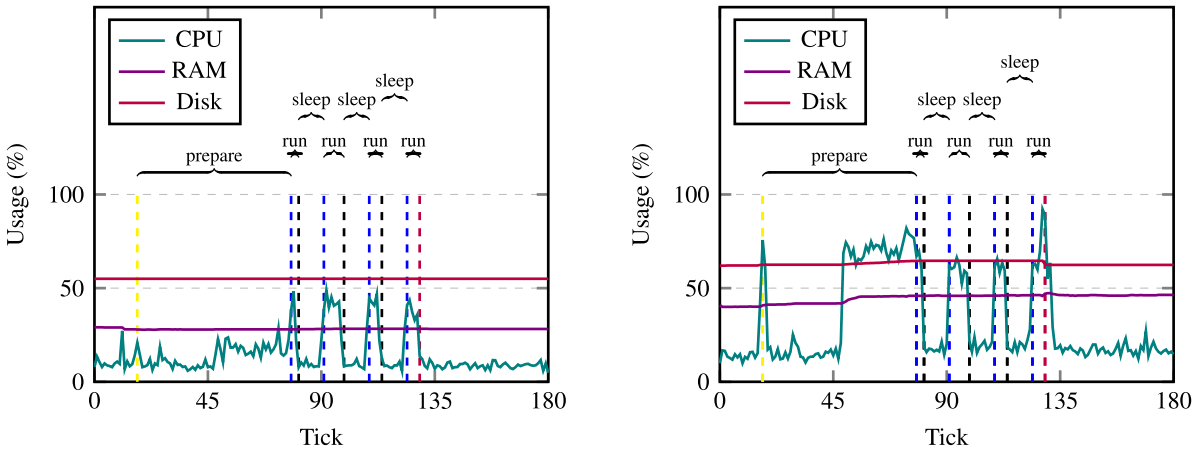
	Metric	Maximum	Average	Std Dev
Node	Logical CPU cores	2	–	–
	Physical CPU cores	2	–	–
	CPU usage	92.2%	39.04%	24.59%
	CPU power consumption	46.30 W	27.71 W	4.95 W
	CPU energy consumption	5015.63 J	–	–
	CPU temperature	63.5 °C	49.55 °C	4.49 °C
	GPU usage	17%	0.90%	2.50%
	GPU power consumption	43.48 W	17.69 W	2.29 W
	GPU energy consumption	3200.99 J	–	–
	GPU temperature	51 °C	49.03 °C	0.22 °C
	Memory usage	47.3%	44.92%	2.17%
	Disk usage	64.6%	63.20%	1.00%
	Up-time	≈ 5.2 min	–	–
	Ports	13	–	–
Container	CPU usage	46.5 s	30.50 s	15.93 s
	Memory usage	≈ 1.5 GB	≈ 1.07 GB	≈ 3.03 MB
	Disk usage	98.3 kB	98.3 kB	0
	Disk limit	25 GB	–	–
	Memory exceeded	0	–	–
	Packets received	39 k	–	–
	Packets dropped	0	–	–
Database (MySQL)	Connections	40	20.93	14.97
	Most connections	5	–	–
	Connections limit	151	–	–
	Refused connections	0	–	–
	#Selects	11.4 k	–	–
	#Updates	6.3 k	–	–
	#Inserts	5.8 k	–	–
	#Deletes	380	–	–
	#Reads	16.2 k	–	–
	#Writes	38.2 k	–	–
	Up-time	275 s	–	–

the cluster and consuming CPU (Fig. 3(a)) while the worker node has a lower CPU usage since it is not actively running the application anymore (Fig. 3(b)). This phase takes approximately 40 ticks.

The results of the data collection show that Monitainer is a helpful tool, which in this case was targeted to collect the majority of the metrics enumerated in Section 3. This collection was used for demonstration purposes. However, its capabilities in terms of configurability and extensibility can be used to cover more complex and diverse systems. For example, through the integration of external tools such as Prometheus exporters, allowing users to monitor and analyze their specific use cases in greater detail and gain deeper insights into their particular system configurations.

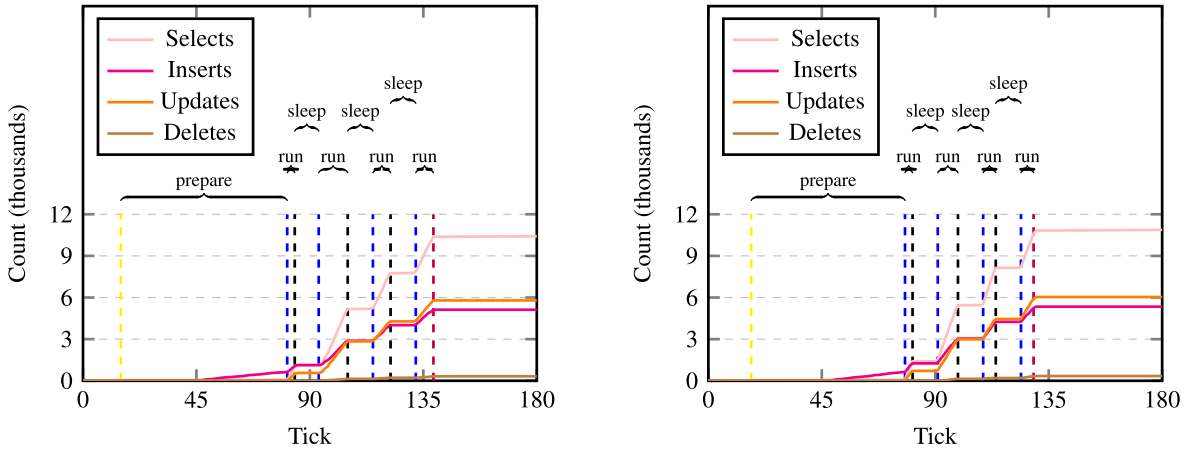
6. Conclusions

This study provides a comprehensive overview of the most widely used containerization and monitoring tools based on an in-depth analysis of numerous research articles in the field. The study aims to shed light on the monitoring challenges posed by container-based systems and present a novel monitoring approach designed to tackle these issues. The Monitainer, described in this paper, allows monitoring nodes, containers, and applications, providing a comprehensive view of the system's behavior. One of the tool's key features is its flexibility and configurability, allowing users to tailor their monitoring to their specific needs. The tool's configuration is based on well-defined templates,



(a) Master node (b) Worker node

Fig. 4. System usage for Swarm.



(a) Kubernetes worker node (b) Swarm worker node

Fig. 5. Application usage.

providing a structured and consistent approach to defining what should be monitored. Time-series data is collected and stored using a storage layer based on Prometheus storage. The metrics collection can be done in various ways, such as once, periodically, or continuously, providing the flexibility to match the system's specific requirements and avoiding extra loads at nodes. Monintainer was designed to be a more transversal monitoring solution, offering support for modern systems based on technologies such as Kubernetes, Docker Swarm, Apache Mesos, and Docker, which allows for a holistic view of containerized systems, reducing the need for distinct tools to monitor multiple containerization platforms and types of metrics. The results from the monitoring can provide insights into the system's behavior, allowing for optimization and resource management decisions based on real-world data.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the European Commission through the Sato Project (Grant agreement ID: 957128), the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020 and the Fundação para a Ciência e a Tecnologia ref. EXPL/CCI-COM/1306/2021.

References

[1] A. Bhardwaj, C.R. Krishna, Virtualization in cloud computing: Moving from hypervisor to containerization—A survey, Arab. J. Sci. Eng. 46 (2021) 8585–8601, http://dx.doi.org/10.1007/s13369-021-05553-3.

[2] S. Newman, Building Microservices, second ed., O'Reilly Media, Inc., 2021.

[3] M. Usman, S. Ferlin, A. Brunstrom, J. Taheri, A survey on observability of distributed edge & container-based microservices, IEEE Access 10 (2022) 86904–86919, http://dx.doi.org/10.1109/ACCESS.2022.3193102.

[4] E. Casalicchio, S. Iannucci, The state-of-the-art in container technologies: Application, orchestration and security, Concurr. Comput.: Pract. Exper. 32 (2020) http://dx.doi.org/10.1002/cpe.5668.

[5] A. Andrae, Total consumer power consumption forecast, Nordic Digit. Bus. Summit 10 (2017) 69.

[6] E. Masanet, A. Shehabi, N. Lei, S. Smith, J. Koomey, Recalibrating global data center energy-use estimates, Science 367 (6481) (2020) 984–986, http://dx.doi.org/10.1126/science.aba3758, URL https://www.science.org/doi/abs/10.1126/science.aba3758, arXiv:https://www.science.org/doi/pdf/10.1126/science.aba3758.

[7] Wavestone relatory, 2021. URL https://theshiftproject.org/wp-content/uploads/2021/03/Note-danalyse_Numerique-et-5G_30-mars-2021.pdf. (Accessed October 2023).

[8] Docker, 2023. URL https://www.docker.com/. (Accessed October 2023).

[9] LXC, 2023. URL https://linuxcontainers.org/. (Accessed October 2023).

[10] rkt, 2023. URL https://github.com/rkt/rkt. (Accessed October 2023).

[11] KVM, 2023. URL https://www.linux-kvm.org/page/Main_Page. (Accessed January 2023).

[12] Kubernetes, 2023. URL https://kubernetes.io/. (Accessed October 2023).

[13] I.M.A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, A. Palopoli, Container orchestration engines: A thorough functional and performance comparison, in: ICC 2019 - 2019 IEEE International Conference on Communications, ICC, 2019, pp. 1–6, http://dx.doi.org/10.1109/ICC.2019.8762053.
[14] D. Swarm, 2023. URL https://docs.docker.com/engine/swarm/. (Accessed October 2023).
[15] Apache Mesos, 2023. URL https://mesos.apache.org/. (Accessed October 2023).
[16] Red Hat OpenShift, 2023. URL https://www.redhat.com/en/technologies/cloud-computing/openshift. (Accessed October 2023).
[17] E. Truyen, M. Bruzek, D. Van Landuyt, B. Lagaisse, W. Joosen, Evaluation of container orchestration systems for deploying and managing NoSQL database clusters, in: 2018 IEEE 11th International Conference on Cloud Computing, CLOUD, 2018, pp. 468–475, http://dx.doi.org/10.1109/CLOUD.2018.00066.
[18] Docker stats, 2023. URL https://docs.docker.com/engine/reference/commandline/stats/. (Accessed October 2023).
[19] cAdvisor, 2023. URL https://github.com/google/cadvisor. (Accessed October 2023).
[20] Prometheus, 2023. URL https://prometheus.io/. (Accessed October 2023).
[21] Sensu, 2023. URL https://sensu.io/. (Accessed October 2023).
[22] Sysdig, 2023. URL https://sysdig.com/. (Accessed October 2023).
[23] F. Moradi, C. Flinta, A. Johnsson, C. Meirosu, ConMon: An automated container based network performance monitoring system, in: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management, IM, 2017, pp. 54–62, http://dx.doi.org/10.23919/INM.2017.7987264.
[24] I. Instana, 2023. URL https://www.instana.com/. (Accessed October 2023).
[25] Dynatrace, 2023. URL https://www.dynatrace.com/. (Accessed October 2023).
[26] Datadog, 2023. URL https://www.datadoghq.com/. (Accessed October 2023).
[27] E. Casalicchio, V. Perciballi, Measuring docker performance: What a mess!!!, in: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, 2017, pp. 11–16, http://dx.doi.org/10.1145/3053600.3053605.
[28] Grafana, 2023. URL https://grafana.com/oss/. (Accessed October 2023).
[29] Open Container Initiative, 2023. URL https://opencontainers.org/. (Accessed October 2023).
[30] J. Enes, R.R. Expósito, J. Touriño, Bdwatchdog: Real-time monitoring and profiling of big data applications and frameworks, Future Gener. Comput. Syst. 87 (2018) 420–437, http://dx.doi.org/10.1016/j.future.2017.12.068, URL https://www.sciencedirect.com/science/article/pii/S0167739X17316096.
[31] D3.js, 2023. URL https://d3js.org/. (Accessed October 2023).
[32] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, P. Merle, Autonomic vertical elasticity of docker containers with elasticdocker, in: 2017 IEEE 10th International Conference on Cloud Computing, CLOUD, 2017, pp. 472–479, http://dx.doi.org/10.1109/CLOUD.2017.67.
[33] E. Casalicchio, V. Perciballi, Auto-scaling of containers: The impact of relative and absolute metrics, in: 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems, FAS*W, 2017, pp. 207–214, http://dx.doi.org/10.1109/FAS-W.2017.149.
[34] M. Großmann, C. Klug, Monitoring container services at the network edge, in: 2017 29th International Teletraffic Congress, Vol. 1, ITC 29, IEEE, 2017, pp. 130–133.
[35] Á. Brandón, M.S. Pérez, J. Montes, A. Sanchez, Fmone: A flexible monitoring solution at the edge, Wirel. Commun. Mob. Comput. 2018 (2018) 1–15.
[36] V. Colombo, A. Tundo, M. Ciavotta, L. Mariani, Towards self-adaptive peer-to-peer monitoring for fog environments, in: Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2022, pp. 156–166.
[37] S. Forti, M. Gaglianese, A. Brogi, Lightweight self-organising distributed monitoring of fog infrastructures, Future Gener. Comput. Syst. 114 (2021) 605–618.
[38] S.F. Piraghaj, A.V. Dastjerdi, R.N. Calheiros, R. Buyya, A framework and algorithm for energy efficient container consolidation in cloud data centers, in: 2015 IEEE International Conference on Data Science and Data Intensive Systems, 2015, pp. 368–375, http://dx.doi.org/10.1109/DSDIS.2015.67.
[39] Yocto-Watt, 2023. URL https://www.yoctopuce.com/EN/products/usb-electrical-sensors/yocto-watt. (Accessed October 2023).
[40] D.-K. Kang, G.-B. Choi, S.-H. Kim, I.-S. Hwang, C.-H. Youn, Workload-aware resource management for energy efficient heterogeneous docker containers, in: 2016 IEEE Region 10 Conference, TENCON, 2016, pp. 2428–2431, http://dx.doi.org/10.1109/TENCON.2016.7848467.
[41] RCE PM600, 2023. URL https://www.rce.it/PM600/PM600.htm. (Accessed October 2023).
[42] S.S. Tadesse, F. Malandrino, C.-F. Chiasserini, Energy consumption measurements in docker, in: 2017 IEEE 41st Annual Computer Software and Applications Conference, Vol. 2, COMPSAC, 2017, pp. 272–273, http://dx.doi.org/10.1109/COMPSAC.2017.117.
[43] H.P. Modular PDU, 2023. URL https://www.hpe.com/psnow/doc/c04123329. (Accessed October 2023).
[44] S.K. Tesfatsion, C. Klein, J. Tordsson, Virtualization techniques compared: Performance, resource, and power usage overheads in clouds, in: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 145–156, http://dx.doi.org/10.1145/3184407.3184414.
[45] E. Santos, C. McLean, C. Solinas, A. Hindle, How does docker affect energy consumption? Evaluating workloads in and out of docker containers, J. Syst. Softw. 146 (2017) http://dx.doi.org/10.1016/j.jss.2018.07.077.
[46] C. Jiang, Y. Wang, D. Ou, Y. Li, J. Zhang, J. Wan, B. Luo, W. Shi, Energy efficiency comparison of hypervisors, Sustain. Comput.: Inform. Syst. 22 (2019) 311–321, http://dx.doi.org/10.1016/j.suscom.2017.09.005, URL https://www.sciencedirect.com/science/article/pii/S2210537917300963.
[47] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, D. Rajwan, Power-management architecture of the intel microarchitecture code-named sandy bridge, IEEE Micro - MICRO 32 (2012) 20–27, http://dx.doi.org/10.1109/MM.2012.12.
[48] R. Shea, H. Wang, J. Liu, Power consumption of virtual machines with network transactions: Measurement and improvements, in: IEEE INFOCOM 2014 - IEEE Conference on Computer Communications, 2014, pp. 1051–1059, http://dx.doi.org/10.1109/INFOCOM.2014.6848035.
[49] A. Asnaghi, M. Ferroni, M. Santambrogio, DockerCap: A software-level power capping orchestrator for docker containers, in: 2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering, DCABES, 2016, pp. 90–97, http://dx.doi.org/10.1109/CSE-EUC-DCABES.2016.166.
[50] R. Brondolin, M. Arnaboldi, T. Sardelli, S. Notargiacomo, M.D. Santambrogio, Energy efficiency for autonomic scalable systems: Research objectives and preliminary results, in: 2018 IEEE 4th International Forum on Research and Technology for Society and Industry, RTSI, 2018, pp. 1–5, http://dx.doi.org/10.1109/RTSI.2018.8548400.
[51] R. Brondolin, T. Sardelli, M.D. Santambrogio, DEEP-mon: Dynamic and energy efficient power monitoring for container-based infrastructures, in: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, 2018, pp. 676–684, http://dx.doi.org/10.1109/IPDPSW.2018.00110.
[52] A. Noureddine, PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools, in: 18th International Conference on Intelligent Environments, Biarritz, France, 2022, URL https://hal.science/hal-03608223.
[53] Marathon, 2023. URL https://mesosphere.github.io/marathon/. (Accessed October 2023).
[54] sysbench, 2023. URL https://github.com/akopytov/sysbench. (Accessed October 2023).
[55] J. Maqbool, S. Oh, G.C. Fox, Evaluating ARM HPC clusters for scientific workloads, Concurr. Comput.: Pract. Exper. 27 (17) (2015) 5390–5410.