The Journal of Systems and Software 208 (2024) 111906

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Monitoring tools for DevOps and microservices: A systematic grey literature review☆

L. Giamattei[a], A. Guerriero[a], R. Pietrantuono[a,*], S. Russo[a], I. Malavolta[b], T. Islam[b], M. Dînga[b], A. Koziolek[c], S. Singh[c], M. Armbruster[c], J.M. Gutierrez-Martinez[d], S. Caro-Alvaro[d], D. Rodriguez[d], S. Weber[e], J. Henss[e], E. Fernandez Vogelin[f], F. Simon Panojo[f]

[a] *University of Naples Federico II, Italy*
[b] *Vrije Universiteit Amsterdam, The Netherlands*
[c] *Kastel - Karlsruhe Institute of Technology, Karlsruhe, Germany*
[d] *University of Alcalá, Alcalá de Henares, Madrid, Spain*
[e] *FZI Research Center for Information Technology, Karlsruhe, Germany*
[f] *Panel Sistemas Informaticos, Madrid, Spain*

ARTICLE INFO

Dataset link: https://github.com/uDEVOPS2020/Monitoring-Tools-for-DevOps-and-Microservices-a-Systematic-Study, https://doi.org/10.5281/zenodo.8212052

Keywords:
Monitoring
Microservice
DevOps
MSA
Tools

ABSTRACT

Microservice-based systems are usually developed according to agile practices like DevOps, which enables rapid and frequent releases to promptly react and adapt to changes. Monitoring is a key enabler for these systems, as they allow to continuously get feedback from the field and support timely and tailored decisions for a quality-driven evolution. In the realm of monitoring tools available for microservices in the DevOps-driven development practice, each with different features, assumptions, and performance, selecting a suitable tool is an as much difficult as impactful task.

This article presents the results of a systematic study of the grey literature we performed to identify, classify and analyze the available monitoring tools for DevOps and microservices. We selected and examined a list of 71 monitoring tools, drawing a map of their characteristics, limitations, assumptions, and open challenges, meant to be useful to both researchers and practitioners working in this area. Results are publicly available and replicable.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

1. Introduction

It is well-known in both academia (Di Francesco et al., 2019) and practice (Waseem et al., 2021) that developing and operating microservice-based systems is a difficult task, mainly due to their distributed nature, complex and dynamic deployments, and technological heterogeneity (Soldani et al., 2018). Having a stable monitoring infrastructure is a strong requirement for operating microservice-based systems where relevant incidents (*e.g.,* faults, performance issues, security breaches) are promptly detected and diagnosed (Waseem et al., 2020).

Various *monitoring tools* are currently widely-used by DevOps teams for collecting, aggregating, and analyzing metrics in order to give meaningful insights about the system's overall health and behavior at runtime. In the context of DevOps, monitoring tools continuously collect system-level metrics (*e.g.,* CPU load, network traffic statistics, failures), aggregate them into higher-level metrics (if needed), and analyze them, primarily with the goal of alerting DevOps teams when a relevant signal is detected, so that they can take corrective actions (Ebert et al., 2016; Hernantes et al., 2015). Representative examples of such monitoring tool include `Prometheus`,[1] `Jaeger`,[2] and `Elasticsearch`.[3]

However, the current landscape of monitoring tools for DevOps and microservices is extremely fragmented, with tens of available tools, each with different goals, monitored entities, produced metrics, technical constraints, underlying technologies, applied monitoring patterns,

☆ Editor: Alexander Serebrenik.
* Corresponding author.
E-mail address: roberto.pietrantuono@unina.it (R. Pietrantuono).
[1] https://prometheus.io
[2] https://www.jaegertracing.io
[3] https://www.elastic.co/elasticsearch

https://doi.org/10.1016/j.jss.2023.111906
Received 9 August 2023; Received in revised form 15 November 2023; Accepted 19 November 2023
Available online 22 November 2023

etc. As an indication, a recent study targeting microservices practitioners identified 23 different monitoring tools used by practitioners when monitoring microservices systems (Waseem et al., 2021). In this context, choosing the right monitoring tool for DevOps and microservices is definitely not trivial and can lead to severe consequences in terms of tool lock-in and systems' quality of service.

The **goal** of this study is to systematically identify, classify, and analyze available monitoring tools for DevOps and microservices. In particular, we are interested in those tools that allow developers to dynamically gather, interpret, and act upon information about a running microservice-based system in the context of DevOps.

The **design of this study** follows the *grey literature review* research method (Rothstein and Hopewell, 2009). We opted for this research method in order to keep the study in scope and well aligned with the objects of analysis (*i.e.,* the tools). Since our study is primarily practitioner-oriented, the objects under analysis are the monitoring *tools*, which are by definition primarily developed and maintained by practitioners (*i.e.,* the tool vendors). Moreover, today it is becoming common practice that Software Engineering practitioners publish various types of grey literature (*e.g.,* tools documentation, white papers, technical reports, blog posts) besides formal academic literature (Garousi et al., 2019) (*e.g.,* conference and journal publications). To the best of our knowledge, this study is the first systematic investigation into the landscape of monitoring tools for DevOps and microservices. The map emerging from this study provides a comprehensive, elaborated, and replicable picture of the current offer of monitoring tools for DevOps and microservices.

The **execution of this study** follows three main phases: (i) search and selection, (ii) data extraction, and (iii) synthesis. Specifically, in the search and selection phase we firstly mined GitHub and the grey literature in search of repositories or web pages mentioning monitoring tools in the context of microservices or DevOps; this initial phase led to a total of 94 potentially-relevant sources. After rigorously applying a set of selection criteria we identified 81 data sources from which we then selected an initial set of 181 potentially-relevant tools. Then, we applied a second set of selection criteria specific to monitoring tools, leading to the final set of 71 tools to analyze. In the data extraction phase, in addition to demographics, four teams of multiple researchers iteratively categorized the 71 tools according to 26 individual parameters organized into three main facets: (i) *general characteristics* (*e.g.,* main goal of the tool, its assumptions, support for visualization, addressed challenges), (ii) *what* the tool monitors (*e.g.,* collected metrics, support for distributed tracing, targeted quality attributes), and (iii) *how* the tool implements the monitoring process (*e.g.,* applied monitoring patterns, required services instrumentation, integration with testing frameworks). Finally, the four teams of researchers quantitatively and qualitatively analyzed the extracted data for all 71 monitoring tools, cross-checked their results (both between and within tools), and synthesized the map of the emerging characteristics of the analyzed tools.

In summary, the **main contributions** of this study are:

- a systematic map of the landscape of available monitoring tools for DevOps and microservices;
- a reusable classification framework for categorizing monitoring tools for DevOps and microservices;
- a discussion about the main implications of the emerging map for both researchers and practitioners working on monitoring methods and techniques for DevOps and microservices;
- a publicly-available archived replication package for the independent verification and replication of this study as *Open Science* material (uDEVOPS2020, 2023).[4]

The **target audience** of this study consists of both researchers and practitioners. Specifically, *researchers* can use our map of 71 monitoring tools to get a detailed overview of the characteristics of existing monitoring tools for DevOps and microservices and use it to either (i) steer their own research towards aspects/characteristics that are still not covered by existing tools or (ii) identify monitoring tools which can be reused as building blocks in their own research on DevOps and microservices (*e.g.,* for identifying which monitoring tool supports distributed tracing and how). Similarly, *DevOps engineers* (i.e., engineers who maintain and monitor a specific system) can use our map of 71 monitoring tools as a decision guidance to identify those that better fit their needs in terms of targeted quality attributes (*e.g.,* performance, security), monitored metrics (CPU usage, network latency, energy consumption), applied monitoring patterns (*e.g.,* health check API, log aggregation), etc. *Tool vendors, i.e.,* those entities maintaining a monitoring tool (*e.g.,* the Cloud Native Computing Foundation, which maintains the tool Prometheus) can use our map of 71 monitoring tools not only to identify competing tools, but also to identify gaps within the monitoring tools landscape and thus anticipate the features of their next generation monitoring tools. Also, researchers and tool vendors can use the emerged classification framework as an instrument for putting their own monitoring tools in context within the current landscape of available monitoring tools for DevOps and microservices; this helps researchers and tools vendors in avoiding to reinvent the wheel and in better understanding what are the differentiating factors of their proposed monitoring techniques, methods, and tools.

The remainder of this paper is structured as follows. Section 2 presents background information on DevOps, microservices, and their monitoring process. Section 3 described the design of this study, while Sections Section 4, 5, 6, and 7 report the obtained results. Section 8 discusses the main implications of the obtained results for researchers and practitioners. Section 9 elaborates on the main threats to the validity of the study. Section 10 discusses related work, while Section 11 closes the paper and discusses future work.

2. Background

2.1. Microservice-based systems

The terms microservice and microservice architecture (MSA) have no single widely accepted definition (Di Francesco et al., 2019). We rely on the popular definition of Fowler and Lewis, which defines a MSA as "*approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API*" (Lewis and Fowler, 2014). They further point out some typical characteristics of MSA, which are "*organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data*" (Lewis and Fowler, 2014).

Many companies like Amazon, Netflix, LinkedIn, Spotify and SoundCloud have shifted the underlying architectural style of their applications to MSA. This enables them to design, develop, test and deploy their software in an agile manner. Continuous delivery of their applications is made possible by automated infrastructure for building, testing and deploying. Decentralizing governance and data management together with the lightweight communication mechanism allows technology independence of the different microservices. They also counter the challenges caused by the distribution of both the software functionality into the microservices and the microservices in cloud infrastructure. These challenges include network latency, possible transmission faults, service orchestration and load balancing (Di Francesco et al., 2019).

By emphasizing the loose coupling and lightweight communication, microservice-based systems foster a high modularity, making the application easier to understand, develop and test, and facilitating the deployment, execution, and maintenance.

[4] https://github.com/uDEVOPS2020/Monitoring-Tools-for-DevOps-and-Microservices-a-Systematic-Study

2.2. DevOps

Due to the above-mentioned desired characteristics, the development and deployment level is often managed by DevOps-like practices. Similar to microservices, the term DevOps has no single definition. Jabbari et al. identified central components of DevOps definitions in their systematic mapping study (Jabbari et al., 2016). The term is a combination of Development and Operations. DevOps enables communication and collaboration and bridges the gap between these two groups resulting in efficient team work. Additionally it unifies methods and tools of software development to respond to the interdependencies between development and operations. Software delivery is achieved through "*continuous feedback, quick response to changes and using automated delivery pipelines resulting in reduced cycle time*" (Jabbari et al., 2016). The deployment process uses source code in version control and automatically deploys to the production environment. DevOps enables continuous integration and the combination of concerns of quality assurance with development and operations. The whole process is supported by cloud technologies (e.g., containers for deployment and cloud platforms at infrastructural level), which greatly alleviate several manual tasks — saving technical stakeholders time to deliver greater value. Ebert et al. (2016) additionally mention that DevOps is an organizational shift to cross-functional teams. In regards to technologies they also name logging and monitoring as important technologies to ensure reliable operation by constantly surveilling the health of software and hardware. They point out that DevOps is suited for cloud and web development and was adopted early on, but less for domains like safety-critical or embedded systems.

Even though Microservices and DevOps originate independently, they share the same set of principles and cultural background, stressing concepts like agility, flexibility, scalability, automation, user-oriented development and cloud-based provisioning. The production paradigm based on microservices and DevOps-like practices is the foundation of many enterprise applications today (Waseem et al., 2021; Di Francesco et al., 2017).

2.3. Monitoring process

For the definition of *monitoring tool*, we rely on the definition provided by Schroeder: A monitoring tool is "*a process or set of possibly distributed processes whose function is the dynamic gathering, interpreting, and acting on information concerning an application as that application executes.*" (Schroeder, 1995) In the context of DevOps as example, monitoring tools collect information at different levels, analyze them, and provide reports with insights into the monitored system or alert DevOps teams (Ebert et al., 2016; Hernantes et al., 2015).

Waseem et al. categorize monitoring tools into libraries and platforms: "*Monitoring libraries are used during the development of microservices and permit collecting the application data. In contrast, monitoring platforms allow gathering and analyzing data from different sources, such as the hosts, infrastructure, and microservices.*" (Waseem et al., 2021) From grey literature, they identified challenges when monitoring microservices and the following practices employed by monitoring tools: log management, exception tracking, health check API, deployment logging, audit logging, and distributed tracking (Waseem et al., 2021). In addition, Richardson describes patterns which can be implemented to monitor microservices (Richardson, 2018). These patterns include the health check API pattern (an endpoint reports the status of a microservice so that it can be called to check the microservice's health), distributed tracing pattern (information of requests between microservices is recorded), application metrics pattern, audit logging pattern (user activity is logged), exception tracking pattern, and log aggregation pattern. These categorizations are considered in our study too.

3. Study design

3.1. Research questions

The goal of the review is to characterize the existing monitoring tools for DevOps and microservices. We formulate the following high-level research questions:

- **RQ1.** *What are the main characteristics of monitoring tools for microservice-based systems?* With this RQ, we will investigate the main functional and technological features of the tools, and analyze if and how the tools address the list of relevant monitoring challenges as identified by Waseem et al. (2021).
- **RQ2.** *What information is gathered to characterize the behavior of the monitored system?* With this RQ, we focus on which metrics, traces and logs the tools is able to extract.
- **RQ3.** *How does the tool implement the monitoring process?* This RQ aims at categorizing the patterns and practices used to gather data as well as its integration with testing.

Each of these questions will be explored with reference to a list of dimensions to characterize the functional and technological features, the gathered metrics, and the way these metrics are gathered, e.g., in terms of monitoring patterns, practices and granularity.

3.2. Tools selection process

To address the above questions, the research is conducted according to the protocol shown in Fig. 1.

The search and selection process considered GitHub as primary source to find relevant monitoring tools ①. However, although GitHub accounts for over 100 million developers and 372 million repositories as of January 2023, we complemented the search via the Google search ② engine to also cover grey literature from which tools and prototype can be made available on the web (e.g., personal, companies or institutions' web pages). In particular, we have:

- searched for *awesome monitoring devops* on GitHub and manually checked all "awesome repositories";
- searched for all repositories on Github having both *monitoring* and *DevOps* as topics of their description;
- searched for all repositories on Github having both *monitoring* and *microservices* as topics of their description;
- searched for *monitoring AND devops* on Google and manually analyzing the first 100 results;
- searched for *monitoring AND microservices* on Google and manually analyzing the first 100 results.

The search process was performed in *August, 2022*.

By merging the results of these searches ③3, we obtained an initial list of 94 sources (e.g., repositories, web pages), where each source may contain more than one tool. These are 7 awesome repositories, 72 sources from searching on GitHub with the topics and 15 Google pages.

We then applied the following inclusion and exclusion criteria to the initial sources:

- Inclusion criteria (sources)
  - IC1: Sources describing at least one monitoring tool that gathers data about the execution of a running system;
  - IC2: Sources discussing either DevOps practices or microservices-based systems;
  - IC3: Sources written in English.
- Exclusion criteria (sources)
  - EC1: Sources whose contents strongly overlap with those of an already-considered source;

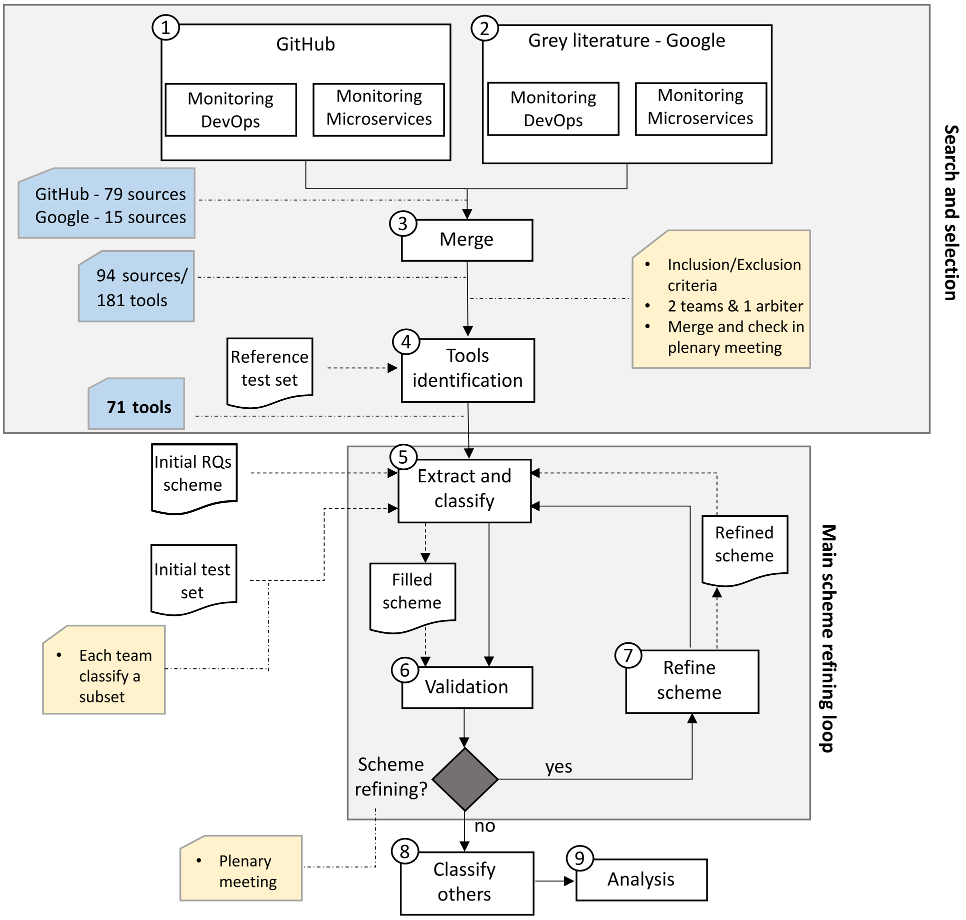


Fig. 1. Search and classification protocol.

- EC2: Entries that are not available, and hence not analyzable (e.g., the link to a web page is broken);
- EC3: Sources that refer exclusively to tools whose primary aim is not monitoring, such as development frameworks, visualization platforms, traces manipulators, etc;
- EC4: Sources that are in the form of scientific publication;
- EC5: Sources reporting exclusively the basic principles of DevOps and microservices, without mentioning any monitoring tool;
- EC6: Videos, podcasts, and webinars since they are too time-consuming to be considered for this phase of the study.

This allowed us to exclude 13 sources resulting in 81 sources (that are: 3 awesome repositories, 6 google pages, and all the 72 repositories coming from the topics search).[5] On this resulting list of sources, we extracted an initial list of 181 tools. Then, we applied the following Inclusion/Exclusion criteria to the list of tools:

- Inclusion criteria (tools)
  - IC1: The tool gathers data about the execution of a running system;
  - IC2: The tool allows monitoring of microservice-based systems and/or DevOps-based processes;
  - IC3: The tool is self-contained, meaning that it does not rely exclusively on being integrated with a 3rd party;
  - IC4: The tool is publicly available (either as an open-source or commercial product);
  - IC5: The documentation of the tool is publicly available and it contains enough information for assessing the use cases, monitoring strategy and to install the system;
  - IC6: The documentation of the tool is in English;
- Exclusion criteria (tools)
  - EC1: Tools whose primary aim is not monitoring, such as development frameworks (e.g., Mortar), visualization platforms (e.g., Graphite), data processing pipelines (e.g., Logstash) etc;
  - EC2: The tool is not accessible, either available for download as a binary that can be run on current operating systems from an official website or an affiliated platform supporting it (e.g., a GitHub repository), or as a SaaS product, centrally hosted;
  - EC3: The tool is explicitly declared as either discontinued, unmaintained, or not yet released.

The process produced a final list of **71 monitoring tools** ④ to analyze, reported in Table 1. Table 2 details, for the two primary sources (i.e., GitHub and Google), the respective resulting tools. Notably, GitHub gave a significantly higher number of tools. Two research teams were involved in applying the criteria independently to the entire set of tools, with the support of one senior researcher as arbiter. The results were then compared and conflicts were solved. We used Cohen's kappa to assess the level of agreement between the raters. This statistic is a good fit in this case, since we only have two raters, both evaluating identical items (the monitoring tools). According to

[5] Most repositories and Google pages contained lists of tools, while the repositories coming from the topic search single tools.

Table 1
Analyzed monitoring tools.

ID	Name	ID	Name
T1	Prometheus	T37	ServerDensity
T2	Zipkin	T38	InsightOps
T3	Apache skywalking	T39	AppSignal
T4	Jaeger	T40	netdata
T5	Nagios enterprise	T41	pyroscope
T6	Zabbix enterprise	T42	gatus
T7	Ganglia	T43	cloudprober
T8	Zenoss enterprise	T44	dd-agent (DataDog)
T9	Opserver	T45	swagger-stats
T10	Icinga	T46	kardia
T11	Naemon	T47	Health
T12	Shinken	T48	WebApiMonitoring
T13	Centreon	T49	terminator
T14	Opsview	T50	MetroFunnel
T15	Check_mk	T51	scope
T16	NSCP	T52	MyPerf4J
T17	collectd	T53	vigil
T18	falcon-plus github	T54	Chronos
T19	fluent-bit	T55	easeagent
T20	influxdata	T56	syros
T21	OpenTSDB	T57	OpenSignals
T22	kairosDB	T58	haystack-client-java
T23	elasticsearch	T59	Monit
T24	javamelody github	T60	Splunk
T25	kamon github	T61	ChaosSearch
T26	Bosun	T62	Sematext
T27	OpenTelemetry	T63	AppDynamics
T28	pinpoint github	T64	Reimann
T29	AWS CloudWatch	T65	Glowroot
T30	StackDriver	T66	GrayLog
T31	Sensu	T67	DataDog
T32	Sentry	T68	Librato (now AppOptics)
T33	CopperEgg	T69	Akamai mPulse
T34	loggly	T70	Sumo Logic
T35	NewRelic	T71	Dynatrace
T36	Papertrail		

Table 2
Source-Tool mapping.

Source	Query	Tools
GitHub	Monitoring DevOps	T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, T17, T18, T19, T20, T21, T22, T23, T24,T25, T26, T27, T28, T29, T30, T31, T32, T33, T34, T35,T36, T37, T38, T39, T40, T41, T42, T43, T44, T45, T62
	Monitoring Microservices	T32, T46, T47, T48, T49, T50, T51, T52, T53, T54, T55, T56, T57, T58
Google	Monitoring DevOps	T1, T5, T6, T31, T32, T35, T59, T60, T61, T62, T63, T67, T68, T69, T70, T71
	Monitoring Microservices	T1, T29, T44, T59, T60, T61, T62, T63, T64, T65, T66

the common interpretation in literature (Fleiss et al., 2003), the results ($k = 0.714$) show a substantial level of agreement. This confirms the reliability of the selection phase. Most of the tools were excluded due to EC1. In particular, the primary reason for applying EC1 to the tools was that they provided only visualization support, without gathering data about the execution of a running system (i.e., IC1). Such tools require supplementation by others that provide them with data. Other tools were excluded by EC3, primarily because they were unmaintained. Only a few tools were discarded due to EC2.

The selected tools have then been compared with a reference *test set*, as suggested by well-known guidelines on secondary studies (Petersen et al., 2015; Kitchenham and Brereton, 2013). To confirm that the list of selected tools is representative enough, it should in fact include the tools in the reference test set. The test set includes the following 5 monitoring tools selected according to the GitHub's stars, denoting their popularity: `Prometheus`, `Netdata`, `Jaeger`, `Zipkin`, and `ELK Stack`. All these tools were included in the list of 71 tools.

3.3. Data extraction

With data extraction, we gather key information about each monitoring tool useful for analysis and classification ⑤. Starting from the research questions (*initial RQs scheme* in Fig. 1), we have defined a scheme with 26 dimensions. The initial scheme was defined in plenary meetings based on authors' previous experience and on the literature on microservices monitoring, e.g., Waseem et al. (2021). To refine and agree on the scheme, we used the *initial test set* approach (Petersen et al., 2015): we initially assigned a same set of 4 tools (`Prometheus`, `Jaeger`, `Zipkin`, and `Apache skywalking`) to each of the 4 involved teams, who independently classified them according to the initial scheme. This preliminary phase ensured that the meaning of dimensions was the same for all the 4 teams. The scheme was iteratively refined in a number of 4 plenary meetings, where we discussed the classifications' findings and finalized the dimensions definition (⑥, ⑦).

We then performed a *horizontal* classification on the whole set of tools ⑧, namely by assigning 18 tools to each team, who independently classified the tools according to all dimensions of the scheme. Finally, in order to ensure a homogeneous classification between the teams, the tools were re-classified *vertically*: each team was assigned a subset of sub-dimensions and classified all the 71 tools for only the assigned dimensions. This allowed refining the scheme further, since it favored the detection of inconsistent classifications performed by the teams on a given dimension during the horizontal classification phase. Dedicated online meetings solved such disagreements. Table 3 reports the identified dimensions and sub-dimensions.

3.4. Analysis

⑨ Data collection and summarization are part of the data analysis process, which aims to comprehend, evaluate, and categorize state-of-the-art monitoring tools. The information for each item extracted are tabulated and visually illustrated. In particular, we analyzed the tools considering the groups of dimensions defined in Table 3. We examine the data that has been extracted to perform both a quantitative and qualitative analysis (Sections 4 and 7). Then, a cross-cutting analysis relating sub-dimensions was also performed in order to highlight interesting patterns in the characteristics of the tools (Section 8).

4. Results – overview

Fig. 2 shows the number of selected (included/excluded) tools by source type.[6] GitHub provided the majority of the tools, but with also the biggest number of exclusions (mainly for lack of documentation - IC5). Out of the analyzed tools, 53/71 (73%) are open source, while 18/71 (25%) are commercial tools.

We looked at the first release date of the tools and we noticed that around 2014–2015 there has been a boost in the number of tools released and that further 24 tools were released since then.

For each tool, we retrieved information about all the release dates, so as to check if the tool is still actively maintained. Specifically, the year of the first and last release available can provide information to analyze longevity and current maintenance activity.

Looking at the tools with last release date on 2022 (data gathered in November, 2022), we notice that 47/71 tools have a release in 2022 (66%), while the last release date of the remaining 25/71 (34%) tools are older (in some cases, such as `NewRelic` last-released in 2014, this can indicate that the tool is no longer maintained). In particular, more than 75% of the tools have a last release in 2021 or 2022.

[6] Note that the summed counts in the image can exceed the numbers in the text since tools can belong to more than one category.

Table 3
The data extraction form of this study.

Category	RQ	Dimensions	Definition	Type/Domain
Metadata	Overview	Id	The internally used ID of the tool	"T"+[numeric]
		Name	The name of the tool	Free text
		Website/Link	Link to the tool	Url
		Provider	Organization/authors that developed the tool	Free text
		Release	The release version the analysis was carried out on	Free text
General characteristics		First release	Date of the first available release	Date
		Last release	Date of the latest available release	Date
		Open source	Whether the tool's sources are available openly	[Yes, No]
	RQ1	Target	The target system to monitor	[Microservices, web services, distributed systems in general]
		Features/motivation	Which high-level features are claimed	Free text
		Available format(s) to export data	List of formats to export monitored data (e.g., JSON, CSV)	Free text
		Visualization	The visualization features offered by the tool	Free text
		Programming language(s)	The language(s) the tool is programmed with	Free text
		Integration/Dependency tools	Required and integrable tools	Free text
		Assumptions	Properties that the monitored system should have	Free text
		Addressed challenges, identified by Waseem et al.	MC1 Collection of monitoring metrics data and logs from containers MC2 Distributed tracing MC3 Many components to monitor (complexity) MC4 Performance monitoring MC5 Analyzing the collected data, MC6 Failure zone detection MC7 Availability of the monitoring tools MC8 Monitoring of application running in containers MC9 Maintaining monitoring infrastructures	[MC1, MC2, MC3, MC4, MC5, MC6, MC7, MC8, MC9]
What is monitored	RQ2	Monitoring metrics (user-oriented)	High level, user-oriented metrics (e.g., failure, health)	Free text
		Monitoring metrics (system-oriented)	Low level, system-oriented metrics (e.g., cpu, memory)	Free text
		Requests tracing	Whether the tool support requests tracing	[Yes, No]
		Events/Failures logging	Whether the tool support event/failures logging	[Yes, No]
		Targeted quality attribute(s)	The quality attribute(s) targeted by the tools	[Performance, Energy, Availability, Reliability, Security]
How is monitored	RQ3	Monitoring patterns	Monitoring patterns implemented by the tool	[Health Check API Pattern, Distributed Tracing Pattern, Application Metrics Pattern, Audit Logging Pattern, Exception Tracking Pattern, Log Aggregation Pattern, Other]
		Monitoring granularity	The granularity of the monitored system	[MSA, microservice, VM/container, infrastructure]
		Monitoring practices	Monitoring practices adopted by the tool	[Log management, exception tracking, health check API, deployment logging, audit logging, distributed tracking]
		Instrumentation	Information about what instrumentation is required	Free text
		Integration with testing	Whether the tool support testing	[Yes, No]

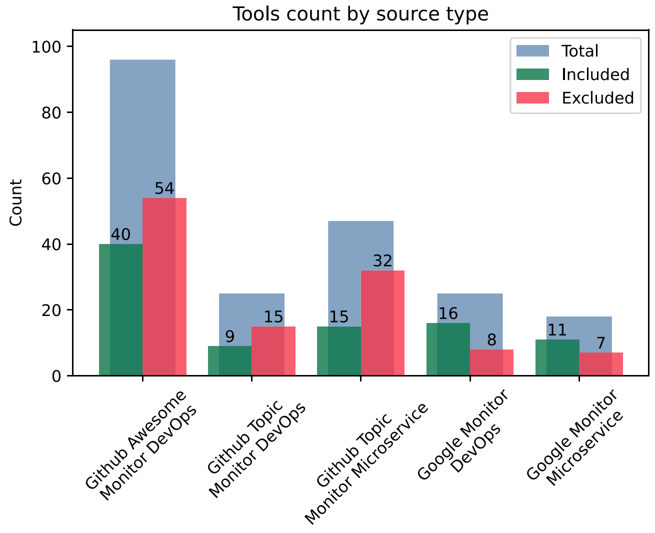


Fig. 2. Tools selection count.

We further analyzed this aspect by combining data about longevity and Open Source, and we noticed that proprietary tools are generally more stable, with a couple of exceptions such as `Zabbix` and `Nagios`, which are Open Source and live since more than 20 years. The longest-living commercial tool is `Splunk`.

5. Results – functional and technological features. Addressed challenges

5.1. Targets, features, motivation

We have analyzed the scope of application of the tools, i.e., the granularity they focus on. Although all the included tools allow monitoring of microservices, we distinguish between tools conceived for monitoring distributed systems in general, tools primarily focused on Web services (as technology to offer services) and tools specifically focused on microservices (as software architecture).

As shown in Fig. 3, more than a half of the tools target distributed systems in general (42 tools, 59%), while web services (14 tools, 21%) and microservices (15, tools, 20%) were found in relatively few case, i.e., in almost a quarter of the tools, each. This is partly explained by the more recent spread of microservices.

Looking at the main features/motivation in Fig. 4, the tools clearly all have collection and monitoring services (in form of traces, logs and

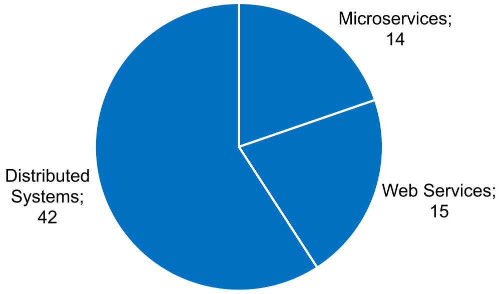


Fig. 3. Tools target.

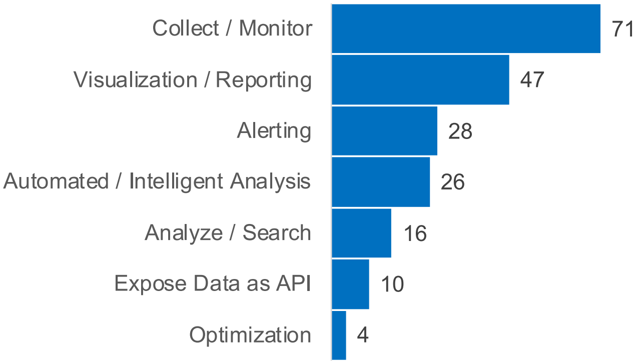


Fig. 4. Tools' features/motivation.

metrics); most of them (66.2%) provide visualization and reporting areas (with visual elements such as panels, graphs or diagrams — more details will follow). Less than 40% have alerting features (e.g., if some thresholds are triggered, alerts are sent to users, in the form of push notifications, emails or web-hooks) and automated analysis (based on collected data, the tools are able to produce and automated, and in some case intelligent, analysis in order to produce data insights). A total of 16 tools (22%) offered the possibility to search under the raw data with custom query languages. 10 tools (14%) can offer their features to external modules (or even to 3rd-party apps) as API endpoints. Finally, 4 tools (≈6%) offer optimization features in order to provide code enhancements to make request processing faster.

The target and features/motivation is the first characteristics to look at when selecting a tool: for instance, while most tools offer some visualization/reporting facility, only few have some form of optimization, and only few expose data as API. These are features that, if needed, will significantly restrict the space of possible choices.

5.2. Reporting

In terms of reporting, the tools have been inspected to determine how the information gathered through monitoring is reported (either for other interacting applications that has to consume that information and/or for the final user). We focus on two main aspects of reporting: *data export format* and *visualization*.

Table 4 reports the tools by data export format category. JSON is the most used format; it is both human-readable and suitable to be easily read from other applications. The second one is CSV, which is one of the most used storage formats. The third category is DB. This includes a batch of possible databases (MySQL, Apache Cassandra, Elasticsearch, RRD, InfluxDB among others). Specifically, the top-DB are: MySQL and Elasticsearch, followed by RRD tool and InfluxDB.

Table 4
Data export formats.

Export format	#Tools (Percentage)	Tools
JSON	49 (69.0%)	T1, T2, T3, T4, T5, T6, T7, T8, T10, T11, T12, T15, T17, T19, T20, T21, T22, T23, T24, T25, T26, T27, T28, T29, T30, T31, T32, T34, T36, T38, T39, T40, T41, T42, T43, T45, T46, T51, T54, T55, T56, T58, T61, T62, T67, T68, T69, T70, T71
CSV	25 (35.2%)	T1, T2, T7, T8, T10, T12, T13, T14, T15, T17, T19, T20, T28, T40, T43, T45, T54, T55, T56, T58, T59, T63, T65, T66, T70
DB	12 (16.9%)	T2, T8, T11, T13, T17, T18, T28, T33, T34, T40, T52, T67
PDF	6 (8.5%)	T10, T12, T13, T14, T15, T60
XML	5 (7.0%)	T6, T12, T15, T24, T63
TXT	5 (7.0%)	T26, T50, T51, T57, T59
not reported	5 (7.0%)	T9, T16, T33, T37, T53
user-defined	4 (5.6%)	T47, T64, T66, T67
log	4 (5.6%)	T34, T38, T48, T55
Excel	3 (4.2%)	T8, T13, T14
Raw	3 (4.2%)	T12, T44, T49
protobuf	3 (4.2%)	T2, T3, T54
S3	2 (2.8%)	T12, T61
Word	2 (2.8%)	T13, T14
HTML	2 (2.8%)	T8, T40
mq	2 (2.8%)	T2, T28
YAML	1 (1.3%)	T6
HTTP	1 (1.3%)	T2
RTF	1 (1.3%)	T14
streams	1 (1.3%)	T8
YML	1 (1.3%)	T20
ODT	1 (1.3%)	T14
H5	1 (1.3%)	T57
TSV	1 (1.3%)	T36
BIN	1 (1.3%)	T27
DF	1 (1.3%)	T57
Powerpoint	1 (1.3%)	T13
Markdown	1 (1.3%)	T40

The last format among the top-4 ones is PDF. Clearly, this is important for the interoperability, useful when gathered data need to be exploited by other applications (e.g., data analytics or decision support systems).

Regarding visualization, 45/71 tools provide dashboards as reporting means to show the monitoring outcomes. Dashboards are defined by Tableau[7] as "a collection of several views, letting you compare a variety of data simultaneously". In particular, diagrams, charts, and tables are usual visualization means to construct a dashboard.

The most common one is *charts*. The usage of tables is also very popular, as they allow easily reporting summary results. Tables are very useful when placed into a dashboard and coupled with graphs to provide additional details. Examples of tools using dashboards, which are very effective for visual analysis and decisions support, are `Zabbik`, `Icinga`, `Zenoss`; others also exploit external tools, like `Prometheus` that can exploit the `Graphana` visualization platform. The detailed breakdown is in Table 5.

5.3. Technologies

5.3.1. Implementation/supported languages

Languages should be considered when choosing a tool, since they impact the possible integration of the monitoring tool with tools of the organization adopting it (e.g., visualization, analytics, recommender systems) and are related to the extensibility, evolvability and maintainability of the tool. Also, they can indirectly impact the monitoring tool performance (e.g., resource consumption, overhead). From our

[7] Tableau is a leading company of data visualization software production (https://www.tableau.com).

Table 5
Data visualization means.

Viz means	#Tools (Percentage)	Tools
Charts	56 (78.8%)	T1, T2, T3, T4, T5, T6, T7, T8, T10, T11, T12, T13, T14, T15, T16, T17, T18, T19, T20, T21, T22, T23, T24, T25, T26, T27, T29, T30, T32, T33, T34, T35, T37, T38, T39, T40, T41, T43, T45, T51, T52, T54, T55, T58, T60, T61, T62, T63, T64, T65, T66, T67, T68, T69, T70, T71
Tables	52 (73.2%)	T1, T3, T5, T6, T7, T8, T10, T11, T12, T13, T15, T16, T19, T20, T21, T22, T23, T24, T25, T26, T29, T30, T32, T33, T34, T35, T37, T38, T40, T41, T42, T43, T45, T46, T47, T52, T53, T55, T58, T59, T60, T61, T62, T63, T64, T65, T66, T67, T68, T69, T70, T71
Dashboard	46 (64.7%)	T1, T5, T6, T7, T8, T10, T11, T12, T13, T15, T16, T18, T19, T20, T21, T22, T23, T25, T26, T29, T30, T31, T32, T33, T34, T35, T37, T38, T40, T41, T43, T45, T52, T55, T58, T60, T61, T62, T63, T64, T66, T67, T68, T69, T70, T71
N/A	9 (12.7%)	T9, T17, T27, T44, T48, T49, T50, T56, T57

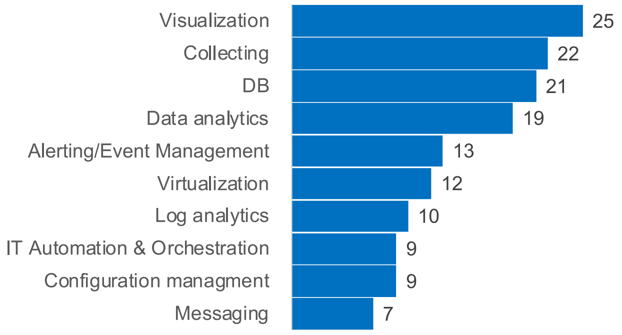


Fig. 5. Tools required technologies.

Table 6
Tools implementation/supported languages.

Language	#Tools (Percentage)	Tools
Go	27 (38.0%)	T1, T3, T4, T6, T8, T18, T20, T23, T26, T27, T28, T29, T30, T31, T32, T35, T41, T42, T43, T44, T49, T51, T56, T67, T68, T70, T71
Java	25 (35.2%)	T2, T3, T8, T21, T22, T23, T24, T27, T28, T30, T32, T35, T38, T50, T52, T55, T57, T58, T62, T63, T65, T66, T70, T71
Python	22 (31.0%)	T3, T5, T7, T8, T12, T14, T15, T16, T18, T22, T23, T27, T28, T30, T32, T34, T37, T38, T60, T63, T70, T71
JavaScript	20 (28.2%)	T3, T4, T6, T13, T15, T22, T23, T27, T34, T36, T37, T38, T45, T46, T51, T56, T62, T66, T68, T69
PHP	14 (19.7%)	T3, T5, T6, T7, T13, T22, T23, T27, T28, T32, T35, T63, T68, T71
C	12 (16.9%)	T5, T6, T7, T11, T16, T17, T19, T28, T35, T40, T59, T63
Ruby	11 (15.5%)	T23, T27, T32, T33, T35, T36, T38, T39, T68, T70, T71
Node.js	10 (14.1%)	T3, T14, T30, T32, T35, T38, T39, T63, T68, T71
C++	9 (12.7%)	T3, T5, T10, T15, T16, T27, T28, T60, T63
.NET	6 (8.5%)	T3, T23, T27, T35, T63, T71
Perl	5 (7.0%)	T5, T7, T14, T17, T23
Rust	4 (5.6%)	T3, T27, T30, T53
Swift	3 (4.2%)	T27, T32, T47
TypeScript	3 (4.2%)	T54, T66, T70
React	3 (4.2%)	T4, T32, T66
XML	2 (2.8%)	T7, T60
C#	2 (2.8%)	T9, T48
Elixir	1 (1.3%)	T39
Escala	1 (1.3%)	T25
Rails	1 (1.3%)	T32
Django	1 (1.3%)	T32
Flashk	1 (1.3%)	T32
Laravel	1 (1.3%)	T32
Scala	1 (1.3%)	T61
Clojure	1 (1.3%)	T64
Erlang	1 (1.3%)	T27

analysis, 27/71 tools report multiple languages in their documentation. In particular, 23/71 are implemented with more than one language, while 4/71 (T22, T28, T30, T36) are implemented mostly in one language, but they support multiple languages. The main reason for the usage/support for multiple languages is the need for agents, sidecars, and/or proxies in different native languages to allow using the monitoring tools in projects developed in various languages (e.g. `Apache Skywalking`, T3). Moreover, the greatest part of the monitoring tools provides user interfaces developed with dedicated languages (e.g. HTML, JavaScript, and so on) (e.g. `Jaeger`, T4). The most used languages are Go and Java, followed by Python and JavaScript. Table 6 reports the detailed results.

5.3.2. Required technologies

Fig. 5 reports the main required technologies for the selected monitoring tools. The integration with visualization tools (like `Grafana`, adopted by T1 and T6, or `Graphite`, adopted by T7 and T10 among others) is fundamental for 35% of the tools (25/71), in line with the results in Table 5.

Collecting is the second type of technology, since the monitoring tools need integration with other tools for data collection during the execution of the system under monitoring. DB technologies are required by approximately 30% of tools for persistence. In addition, many monitoring tools exploit alerting/event management technologies, as they need to capture data in real-time from event sources (like other services or devices), e.g., T4 with `Kafka`. Clearly, the number and type of required technologies can also affect the easiness of adopting and integrating a tool in the organization.

5.3.3. Assumptions

Table 7 reports the assumptions stated for the selected tools, which often come in the form of requirements. For instance, the most common assumption is about the operating system and libraries needed to install and make the tool work. A bunch of tools (7/71) have less stringent assumptions and support several systems. Other assumptions regard the way in which tools are executed. For instance, they regard the need for agents and instrumentation. Clearly, more assumptions required restrict the freedom of selection, as they could be not easily satisfiable. For instance, tracing by `Monit` (T62) supports only Java-based applications, or `Centreon` that officially supports only MariaDB; these can both be quite limiting for microservice-based systems. The full list is in Table 7.

5.4. Addressed challenges

We analyze if and how the tools address the list of challenges as identified by Waseem et al. (2021). The challenges are summarized in Table 8.

The most important challenges addressed by the tools, as shown in Fig. 6, are related to the ability of effectively monitor performance (MC4), to the collection of monitoring metrics and logs from containers (MC1), to deal with complexity (MC3) and to analyze data (MC5). Some of the reasons for the presence of the above challenges are explained by some interviewees in the work by Waseem et al. (2021), and are related to (i) the communication between hundreds of microservices (hence referring to complexity of MSA), (ii) absence of a standardized infrastructure for run-time monitoring (hindering collection of data),

Table 7
Assumptions made by tools.

Assumption	#Tools (Percentage)	Tools
Require agent	13 (18.3%)	T5, T8, T12, T21, T28, T31, T62, T63, T65, T67, T68, T70, T71
Require specific OS	11 (15.5%)	T16, T17, T19, T23, T30, T31, T35, T38, T47, T56, T59
Most of the systems are supported	7 (9.9%)	T3, T22, T26, T29, T32, T34, T36
Require instrumentation	6 (8.5%)	T2, T4, T27, T65, T70, T71
Require connection to backend	5 (7.0%)	T67, T68, T69, T70, T71
Run as SaaS	5 (7.0%)	T33, T34, T37, T62, T63
Require Docker	4 (5.6%)	T51, T55, T56, T58
Require JVM	3 (4.2%)	T25, T52, T65
Require NodeJS	2 (2.8%)	T45, T54
Require OpenTracing version compatibility	1 (1.4%)	T58
Tracing only supports Java-based applications	1 (1.4%)	T62
Require external data storage	1 (1.4%)	T61
Haystack client java needs OpenTracing version	1 (1.4%)	T58
Cross-platform	1 (1.4%)	T1
Require code rebuilding	1 (1.4%)	T54
Maintained in your own system	1 (1.4%)	T53
Require Swift binaries v4.1.2	1 (1.4%)	T47
Require plugin	1 (1.4%)	T24
Require specific DB	1 (1.4%)	T13
Require visualizer	1 (1.4%)	T11
Require script not blocked by browser	1 (1.4%)	T69

Table 8
Challenges addressed, identified in Waseem et al. (2021).

Acronym	Description
MC1	Collection of monitoring metrics data and logs from containers
MC2	Distributed tracing
MC3	Many components to monitor (complexity)
MC4	Performance monitoring
MC5	Analyzing the collected data
MC6	Failure zone detection
MC7	Availability of the monitoring tools
MC8	Monitoring of application running in containers
MC9	Maintaining monitoring infrastructures

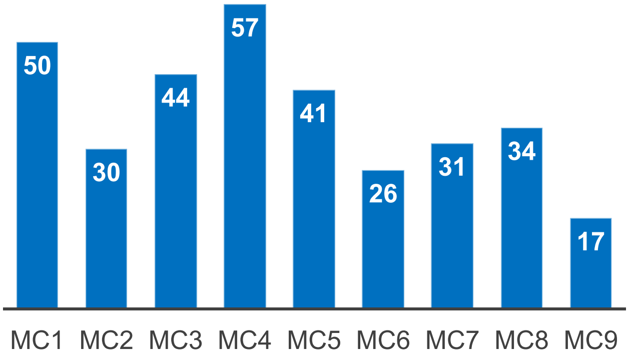


Fig. 6. Challenges addressed.

(iii) different languages, databases, and frameworks for developing microservices, and (iv) logs and dataflows in different format.

Indeed, we found that several of the analyzed tools focus on solutions to (i) efficiently extracting information (useful for service administrators, managers and stakeholders), (ii) dealing with several metrics (e.g., request duration, errors, availability time, database metrics, among others), with performance monitoring (MC4, 57 tools), and with logs (i.e., traces of events and errors), overcoming the issue heterogeneous log formats (MC1, 50 tools). Moreover, the solution that many tools offer is thought to scale, as many of them (MC3, more than 44 tools) are able to work with complex (i.e., with many service and microservice components) architectures.

Other challenges, such as monitoring of application running inside containers are only partially addressed, by half of the tools (MC8, 34 tools), or still mostly neglected (e.g., maintenance of the monitoring infrastructure, MC9, 17 tools); thus suggesting future directions for researchers and tool vendors – a deeper discussion will follow in Section 8.

6. Results – What is monitored

In this section, we report information about *what is monitored* by the analyzed tools. The monitoring tools collect and monitor a wide range of metrics, both at the system-level and user-level. Therefore, we focus on two types of monitoring metrics: *user-oriented metrics* and *system-oriented metrics*. We also extract information regarding the target quality attribute, support for distributed tracing, and failure/events logging for each tool.

6.1. User-oriented metrics

In the context of this study, a user-oriented metric is a *high-level* metric whose values might influence how users of the system experience or perceive the value of the system. Examples of user-oriented metrics include: response time, latency, number of failures/errors, SLA violations, number of network requests, etc.

In terms of user metrics, several tools, 26 of 71 (36.62%) support *User-defined* (*e.g.,* custom events, custom application metrics, etc.) and *Failure* metrics (*e.g.,* error rates, SLA metrics — number of errors/exceptions/failures, etc.). *Timing* (*e.g.,* response time, latency, etc.) and *Networking* (*e.g.,* request rate/error/duration, average in/outbound packets, average packet loss, etc.) related metrics are also popular user-oriented metrics among the tools, which we found in 24 of 71 (33.80%) and 23 of 71 (32.39%) tools, respectively (see Table 9).

For instance, during the data extraction phase, we observed that `Sumo Logic` (T70) collects *Timing*, *Networking*, *Failure*, *UX*, and *User-defined* metrics. Similar to `Sumo Logic`, `Elastic search` (T23) collects *Failure* and *Networking* metrics. Besides, Elastic Search also covers *Health* and *User-sessions* metrics.

6.2. System-oriented metrics

In the context of this study, a system-oriented metric is a *low-level* metric whose values are of interest to DevOps engineers. Those metrics are generally defined at a lower level of granularity (*e.g.,* container, OS, physical node) than user-oriented metrics. Examples of system-oriented metrics include: CPU usage, memory usage, available resources at the OS level, number of database connections, etc.

In terms of system metrics, *Networking*, *Memory*, and *CPU* are the most dominant metrics in more than 70% of the tools. As shown in Table 10, 55 out of 71 tools (77.46%) collect *Networking* metrics (*e.g.,* RX/TX network traffic, average in/outbound packets, average packet loss, etc.), followed by *Memory* (*e.g.,* memory usage, system load, swap, etc.) – 53 of 71 tools (74.65%) – and *CPU* (*e.g.,* CPU usage, host/process/user CPU, thread count, etc.) – 50 of 71 tools (70.42%).

For instance, we observed that `Sumo Logic` (T70) collects *CPU*, *Memory*, *Networking*, *IO*, *DB*, *Failure*, *Timing* system-oriented metrics. Similarly, `Elastic search` (T23) also focuses on *CPU*, *Memory*, *Networking*, *IO*, *DB*, and *Timing* metrics. Besides, `Elastic search` collects metrics related to *Health* and *Failure* (*e.g.,* number of errors/exceptions/failures, etc.).

Table 9
The user-oriented metrics supported by the monitoring tools.

Metric	#Tools (Percentage)	Tools
User-defined	26 (36.62%)	T1, T7, T11, T12, T13, T15, T20, T21, T26, T27, T31, T39, T40, T44, T45, T47, T60, T61, T62, T63, T64, T66, T67, T68, T69, T71
Failure	26 (36.62%)	T1, T6, T8, T10, T14, T15, T17, T19, T23, T24, T26, T29, T30, T32, T33, T35, T40, T44, T45, T59, T60, T63, T67, T68, T70, T71
Timing	24 (33.80%)	T1, T10, T14, T15, T19, T22, T24, T29, T30, T32, T33, T39, T40, T43, T44, T45, T60, T62, T63, T64, T67, T68, T70, T71
Networking	23 (32.39%)	T1, T4, T10, T13, T17, T19, T22, T23, T24, T30, T32, T33, T40, T43, T44, T45, T62, T63, T67, T68, T69, T70, T71
Health	10 (14.08%)	T5, T10, T14, T23, T26, T30, T32, T35, T36, T63
UX	8 (11.27%)	T32, T35, T60, T62, T63, T69, T70, T71
User-sessions	8 (11.27%)	T15, T17, T23, T24, T29, T30, T62, T63
DB	4 (5.63%)	T13, T22, T24, T33
Memory	4 (5.63%)	T29, T33, T44, T62
Application-level metrics	3 (4.23%)	T63, T69, T71
Power	2 (2.82%)	T13, T63
CPU	2 (2.82%)	T33, T44
IO	1 (1.41%)	T10
Success	1 (1.41%)	T43
Profiling	1 (1.41%)	T62
Container-lifecycle	1 (1.41%)	T62
N/A	20 (28.17%)	T16, T18, T28, T34, T37, T38, T41, T42, T46, T48, T49, T50, T51, T52, T53, T54, T55, T56, T57, T58

It is worth noting that system-oriented metrics are much better supported than user-oriented metrics, indirectly denoting that a system-level perspective (in terms, for instance, of resource consumption), which can be useful for capacity and deployment planning is deemed a more important goal than application-level and user-oriented metrics such as response time or latency.

6.3. Distributed tracing

In this section, we report our insights regarding the support for distributed tracing of the tools.

As shown in Table 11, the majority of the tools, 41 of 71 (57.75%), do not support distributed tracing, while 30 tools (42.25%) provide dedicated support for distributed tracing.

These 41 tools include Splunk (T60) which does not have any support for distributed tracing. Whereas, there are tools, such as Jaeger (T4) or Dynatrace (T71) which provide support for distributed tracing requests.

6.4. Failures/events logging

This parameter investigates whether the tools support failures/events logging or not. As shown in Table 12, the majority of the tools, 54 of 71 (76.06%), support failures/events logging, while the remaining 17 tools (23.94%) do not support it.

Among these 54 tools, for instance, Sumo Logic (T70) or Elastic search (T23), provide support for events/failures logging. However, there are some tools that do not provide support for collecting such logs. OpenTelemetry (T27) is one of the 17 tools (23.94%) that does not have support for failures/events logging.

Table 10
System-oriented metrics supported by the tools.

Metric	#Tools (Percentage)	Tools
Networking	55 (77.46%)	T1, T2, T3, T5, T6, T7, T8, T10, T11, T12, T13, T14, T15, T17, T18, T19, T20, T21, T22, T23, T24, T25, T26, T28, T29, T30, T32, T33, T34, T35, T36, T37, T38, T39, T40, T44, T45, T51, T52, T53, T54, T55, T56, T58, T59, T60, T62, T63, T64, T65, T67, T68, T69, T70, T71
Memory	53 (74.65%)	T1, T3, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, T17, T18, T19, T20, T21, T22, T23, T24, T25, T26, T28, T29, T30, T31, T33, T35, T37, T38, T39, T40, T41, T44, T45, T46, T51, T52, T53, T54, T55, T56, T59, T60, T62, T63, T64, T65, T67, T68, T70
CPU	50 (70.42%)	T1, T3, T5, T6, T7, T8, T9, T10, T12, T13, T14, T15, T16, T17, T18, T19, T20, T21, T23, T24, T25, T26, T28, T29, T30, T31, T33, T35, T37, T38, T39, T40, T41, T42, T44, T45, T46, T51, T52, T53, T59, T60, T62, T63, T64, T65, T67, T68, T70, T71
IO	44 (61.97%)	T1, T3, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, T17, T18, T19, T20, T21, T22, T23, T25, T26, T28, T29, T30, T31, T33, T35, T37, T38, T39, T40, T44, T45, T52, T59, T60, T62, T63, T64, T67, T70, T71
Timing	39 (54.93%)	T1, T2, T3, T5, T7, T10, T12, T14, T17, T19, T20, T22, T23, T24, T25, T26, T31, T32, T33, T34, T35, T36, T39, T40, T41, T42, T45, T46, T51, T52, T54, T55, T60, T62, T64, T65, T67, T69, T70
DB	21 (29.58%)	T1, T3, T5, T12, T14, T17, T22, T23, T24, T33, T39, T40, T54, T55, T56, T63, T65, T67, T68, T70, T71
User-defined	17 (23.94%)	T3, T27, T29, T33, T34, T40, T44, T46, T47, T60, T61, T62, T63, T64, T66, T67, T68
Health	16 (22.54%)	T1, T3, T5, T6, T10, T14, T22, T23, T31, T33, T35, T40, T46, T53, T58, T71
Failure	16 (22.54%)	T2, T9, T25, T33, T34, T35, T36, T39, T42, T55, T58, T60, T63, T65, T69, T70
Temperature	6 (8.45%)	T13, T17, T19, T20, T21, T54
Container-lifecycle	5 (7.04%)	T35, T51, T54, T56, T58
Service	4 (5.63%)	T3, T31, T54, T68
Power	4 (5.63%)	T1, T6, T10, T59
Application-level metrics	4 (5.63%)	T24, T41, T52, T55
Load	1 (1.41%)	T26
Process	1 (1.41%)	T71
N/A	6 (8.45%)	T4, T43, T48, T49, T50, T57

6.5. Targeted quality attribute

This parameter is about the quality attributes explicitly targeted by the monitoring tools. As shown in Table 13, the vast majority of the tools – 63/71 (88.73%) – focus on *Performance* as their targeted quality attribute, followed by *Reliability* (53/71, 74.65%). This means that the studied tools tend to focus more on the *Performance* and *Reliability* quality aspects compared to other quality attributes. This result is expected since the performance and reliability of microservice-based systems directly impact the user experience, potentially impacting the most

Table 11
Support for distributed tracing.

Request tracing	#Tools (Percentage)	Tools
Yes	30 (42.25%)	T1, T2, T3, T4, T5, T8, T22, T23, T25, T27, T28, T29, T30, T32, T34, T35, T41, T44, T45, T48, T54, T55, T58, T62, T63, T65, T67, T68, T70, T71
No	41 (57.75%)	T6, T7, T9, T10, T11, T12, T13, T14, T15, T16, T17, T18, T19, T20, T21, T24, T26, T31, T33, T36, T37, T38, T39, T40, T42, T43, T46, T47, T49, T50, T51, T52, T53, T56, T57, T59, T60, T61, T64, T66, T69

Table 12
Support for failure/events logging.

Failure/event logging	#Tools (Percentage)	Tools
Yes	54 (76.06%)	T1, T2, T3, T4, T5, T6, T8, T9, T10, T11, T12, T13, T14, T15, T16, T17, T18, T19, T20, T21, T23, T24, T25, T26, T28, T29, T30, T31, T32, T33, T34, T35, T36, T38, T39, T40, T41, T44, T45, T51, T53, T58, T59, T60, T62, T63, T64, T65, T66, T67, T68, T69, T70, T71
No	17 (23.94%)	T7, T22, T27, T37, T42, T43, T46, T47, T48, T49, T50, T52, T54, T55, T56, T57, T61

Table 13
The quality attributes targeted by the monitoring tools.

Quality attribute	#Tools (Percentage)	Tools
Performance	63 (88.73%)	T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, T17, T18, T19, T20, T21, T22, T23, T24, T25, T26, T28, T29, T30, T31, T32, T33, T34, T35, T36, T37, T39, T40, T41, T43, T44, T45, T46, T48, T51, T52, T54, T55, T56, T58, T59, T60, T61, T62, T63, T64, T65, T66, T67, T68, T69, T70, T71
Reliability	53 (74.65%)	T1, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T17, T18, T19, T22, T23, T24, T25, T26, T28, T29, T30, T31, T33, T34, T36, T37, T38, T39, T40, T41, T42, T43, T46, T48, T51, T52, T53, T56, T57, T58, T59, T60, T61, T62, T63, T67, T68, T70, T71
Security	15 (21.13%)	T5, T6, T14, T19, T35, T38, T40, T51, T60, T61, T66, T67, T69, T70, T71
Usability	7 (9.86%)	T5, T60, T62, T63, T67, T69, T70
User-defined	4 (5.63%)	T27, T47, T64, T66
Energy	3 (4.23%)	T1, T13, T30
None	2 (2.81%)	T49, T50
Maintainability	1 (1.41%)	T1
Compatibility	1 (1.41%)	T5

user acceptance (and the success of the system as a whole). We also observed that *Energy* (3/71, 4.23%), *Maintainability* (1/71, 1.41%), and *Compatibility* (1/71, 1.41%) are the least frequently targeted quality attributes. Four monitoring tools (5.63%) are targeting *User-defined* metrics, *i.e.,* they allow system maintainers to define their our quality-related metrics and provide means to instrument the application in order to suitably log and aggregate such custom metrics; interestingly, two of those monitoring tools (*i.e.,* T27 and T47) support exclusively *User-defined* metrics, meaning that they do not come with predefined quality metrics that system maintainers can use out of the box. Finally, two monitoring tools (2/71, 2.81%) do not explicitly target any quality attribute (not even those defined by system maintainers); in both cases the monitoring tool provides features for collecting and filtering system logs, while delegating to other third-party tools the aggregation of the collected logs into suitable quality metrics.

Table 14
Co-occurring quality attributes targeted by the monitoring tools.

Combination	#Tools (Percentage)	Tools
Performance - Reliability	49 (69.01%)	T1, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T17, T18, T19, T22, T23, T24, T25, T26, T28, T29, T30, T31, T33, T34, T36, T37, T39, T40, T41, T43, T46, T48, T51, T52, T56, T58, T59, T60, T61, T62, T63, T67, T68, T70, T71
Performance - Security	14 (19.72%)	T5, T6, T14, T19, T35, T40, T51, T60, T61, T66, T67, T69, T70, T71
Security - Reliability	12 (16.90%)	T5, T6, T14, T19, T38, T40, T51, T60, T61, T67, T70, T71
Performance - Reliability - Security	11 (15.49%)	T5, T6, T14, T19, T40, T51, T60, T61, T67, T70, T71

For instance, the targeted quality attributes for the `Sumo Logic` tool (T70) are *Performance*, *Reliability*, *Security*, and *Usability*, whereas, `Elastic search` (T23) focuses on *Performance* and *Reliability*.

For the targeted quality attributes, it would be interesting also to note down further insights on their co-existence with other quality attributes. From the collected data, we noticed in 11 cases, 11 of 71 tools, cover *Performance*, *Reliability*, and *Security* together. 49, 14, and 12 tools cover the combination of *Performance-Reliability*, *Performance-Security*, and *Security-Reliability*, respectively, as their targeted quality attributes with/without other less frequent quality attributes. This information related to the combination of the quality attributes is presented in Table 14. It is finally interesting to note that monitoring *Energy* is scarcely supported (3/71), although power consumption is one of the few attributes directly related to cost. We expect an increased support in the near future. *Security* will also likely see an increasing support, considering the pressing need for cyber-security in todays' systems.

7. Results – How is monitoring done

In this section, we discuss how the monitoring is done by the tools. This includes the instrumentation usable with the tools, the monitoring patterns and practices we observed, the granularity on which data is gathered and whether there is an integration with testing.

7.1. Instrumentation

We report our insights regarding the instrumentation used by the monitoring tools in this section. The possible forms of instrumentation are platform, library or no instrumentation. A platform runs besides the monitored application and forwards monitoring data to a back-end for storage and analysis. This data can be either gathered by the platform itself, through automatic instrumentation or by instrumentation libraries. These libraries are programming language specific and enable the manual instrumentation of applications. Platforms and libraries can either be vendor-provided or third-party. No instrumentation means that there is neither a platform nor a library. Instead monitoring data is gathered through manual instrumentation and manual forwarding to the back-end via communication protocols, e.g. a REST-API.

Table 16 shows that the majority of tools, 59 (83.1%), provide a vendor-specific platform. About half of the tools, 37 (52.11%), provide a vendor-specific library and 32 tools provide a vendor-specific platform and library. Third-party platforms are usable with 36 (50.7%) and third-party libraries with 36 (50.7%) of the tools. Only 3 (4.23%) tools provide no instrumentation platform or library or can be used with third-party platforms or libraries. For instance, `OpenTelemtry` (T27) provides platforms and libraries, supports forwarding data to third-party platforms and back-ends and can ingest data from third-party platforms or libraries.

Table 15
Monitoring Patterns. P1: Health Check API, P2: Distributed Tracing, P3: Application Metrics, P4: Audit Logging, P5: Exception Tracking, P6: Log Aggregation.

Monitoring pattern	#Tools (Percentage)	Tools
P1	22 (30.98%)	T42, T43, T46, T53, T55, T56, T5, T6, T33, T10, T11, T19, T29, T35, T58, T31, T62, T39, T45, T51, T52, T54
P2	17 (23.94%)	T2, T4, T50, T5, T8, T23, T25, T32, T27, T44, T31, T62, T39, T45, T51, T54, T65
P3	33 (46.47%)	T1, T7, T24, T47, T57, T59, T5, T6, T33, T8, T10, T13, T15, T36, T22, T23, T25, T32, T26, T38, T64, T27, T44, T29, T35, T58, T31, T62, T39, T45, T51, T52, T65
P4	14 (19.71%)	T23, T5, T6, T33, T8, T13, T15, T36, T26, T38, T64, T31, T62, T37
P5	12 (16.90%)	T5, T23, T26, T38, T64, T37, T39, T45, T51, T52, T65, T66
P6	28 (39.43%)	T14, T18, T20 ,T21, T41, T48, T49, T61, T5, T6, T33, T8, T10, T11, T19, T13, T15, T36, T22, T23, T26, T38, T64, T27, T44, T31, T62, T66
All	11 (15.49%)	T3, T28, T30, T34, T40, T60, T63, T67, T68, T70, T71
N/A	5 (7.04%)	T9, T12, T16 ,T17, T69

Table 16
Instrumentation mechanisms provided by the tools.

Instrumentation	#Tools (Percentage)	Tools
Vendor-provided platform	59 (83.1%)	T1-T18, T21, T23-T41, T43-T44, T46, T50-T56, T58-T60, T62-T63, T65-T68, T70-T71
Vendor-provided library	37 (52.11%)	T1-T3, T6, T15, T17, T22-T29, T32-T35, T37, T39-T41, T44-T48, T53-T54, T60, T63, T65, T67-T71
Third-party platform	36 (50.7%)	T1-T5, T11-T12, T14-T16, T19-T23, T26-T27, T29, T31-T32, T34-T35, T37, T39, T41, T44, T58, T60-T64, T66-T67, T70-T71
Third-party library	36 (50.7%)	T1-T5, T8, T11-T12, T14-T16, T19-T23, T26-T27, T29, T31-T32, T34-T35, T37, T39, T41, T44, T58, T60-T64, T67, T70-T71
No instrumentation	3 (4.23%)	T42, T49, T57

7.2. Monitoring patterns and practices

Table 15 reports the tools as per different monitoring patterns. The monitoring patterns of the tools are classified into 6 categories. P1: Health Check API Pattern, P2: Distributed Tracing Pattern, P3: Application Metrics Pattern, P4: Audit Logging Pattern, P5: Exception Tracking Pattern, P6: Log Aggregation Pattern. For each monitoring pattern, there are specific monitoring practices. For example, if a tool has only monitoring pattern P1, then it only supports health check API and sometimes event logger. In case of P3, there are tools like T47 that supports event logger and T57 that supports signaling theory, stigmergy, systems thinking, semiotics, and social cognition practices. Out of 71, only 11 tools follow all the six monitoring patterns. The overview of the classification is in Table 15. We can see that the analyzed tools support most of the available patterns such as: Application metrics (P3), health check API (P1), log aggregation (P6), distributed tracking (P2).

Most of the tools provide "log management" as monitoring patterns. However, there exists no tools which only supports P4, P5. They are always combined with other monitoring patterns. For 6 tools, no information is available, or it is hard to gather the information.

Table 17
Monitoring granularity mapping tools to every granularity level.

Monitoring granularity	#Tools (Percentage)	Tools
Application	37 (52.11%)	T1, T3, T5, T6, T8, T13, T14, T15, T20, T21, T23, T24, T27, T28, T29, T30, T31, T32, T33, T34, T35, T36, T37, T38, T40, T60, T61, T62, T63, T64, T65, T66, T67, T68, T69, T70, T71
Microservice	62 (87.32%)	T1, T2, T3, T4, T5, T6, T7, T8, T10, T13, T14, T15, T20, T21, T22, T23, T24, T25, T26, T27, T28, T29, T30, T31, T32, T33, T34, T35, T36, T37, T38, T40, T41, T42, T43, T44, T45, T46, T47, T48, T49, T50, T51, T52, T53, T54, T55, T56, T57, T58, T59, T60, T61, T62, T63, T64, T65, T66, T67, T68, T70, T71
VM/Container	42 (59.15%)	T1, T3, T5, T6, T8, T10, T13, T14, T15, T17, T18, T19, T20, T21, T22, T23, T24, T25, T28, T29, T30, T31, T33, T34, T35, T36, T37, T38, T39, T40, T44, T60, T61, T62, T63, T64, T65, T66, T67, T68, T70, T71
Infrastructure	40 (56.33%)	T1, T3, T5, T6, T7, T8, T10, T13, T14, T15, T16, T17, T18, T19, T20, T21, T23, T25, T26, T28, T29, T30, T31, T33, T34, T35, T36, T37, T38, T40, T44, T60, T61, T62, T63, T64, T67, T68, T70, T71
N/A	3 (4.23%)	T9, T11, T12

7.3. Monitoring granularity

Within the monitoring granularity dimension, we investigated on which levels the tools operate. These levels consist of the complete Microservice-based application, individual Microservices, a VM or container, and the infrastructure. The mapping of tools to the levels is displayed in Table 17. While the application, VM/container, and infrastructure levels are supported by over half of the tools, 62 tools (87.32%) target the Microservice level. In addition, Table 18 contains all level combinations which are supported by at least one tool, and the corresponding tools. Here, tools that work at only one among the application, infrastructure level or VM/Container levels are 1 per each level (T69, `Akamai mPulse`, T16, `NSCP`, and T39, `AppSignal`, respectively) while there are 20 tools (28.17%) supporting only *Microservices*. The most frequent combination with 31 tools (42.66%) covers all four levels (application, Microservice, VM/container, and infrastructure level).

As an example, these 31 tools include `Zenoss` which collects metrics and events of the infrastructure, VM/container, and individual service level. By combining these information and providing overviews for complete applications, `Zenoss` is one of the tools that operate on all levels.

7.4. Integration with testing

Regarding testing, we looked if the monitoring tools support testing in some way. As shown in Table 19, only 11 tools (15.5%) provide such support. As a consequence, a majority of the tools do not support tests. When collecting data about integration with testing, we observed that most of them do not inherently support testing in their standard version but rely on plugins for this purpose. Additionally, we noted

Table 18
Monitoring granularity showing which tools support which level combination.

Combinations of monitoring granularity	#Tools (Percentage)	Tools
Application	1 (1.40%)	T69
Application, Microservice	2 (2.81%)	T27, T32
Application, Microservice, VM/Container	3 (4.23%)	T24, T65, T66
Application, Microservice, VM/Container, Infrastructure	31 (43.66%)	T1, T3, T5, T6, T8, T13, T14, T15, T20, T21, T23, T28, T29, T30, T31, T33, T34, T35, T36, T37, T38, T40, T60, T61, T62, T63, T64, T67, T68, T70, T71
Microservice	20 (28.17%)	T2, T4, T41, T42, T43, T45, T46, T47, T48, T49, T50, T51, T52, T53, T54, T55, T56, T57, T58, T59
Microservice, Infrastructure	2 (2.82%)	T7, T26
Microservice, VM/Container	1 (1.40%)	T22
Microservice,VM/Container, Infrastructure	3 (4.23%)	T10, T25, T44
Infrastructure	1 (1.40%)	T16
VM/Container	1 (1.40%)	T39
VM/Container, Infrastructure	3 (4.23%)	T17, T18, T19
N/A	3 (4.23%)	T9, T11, T12

Table 19
Integration with testing.

Integration with testing?	#Tools (Percentage)	Tools
Yes	11 (15.5%)	T24, T29, T56, T58, T59, T60, T62, T63, T64, T67, T69
No	60 (84.5%)	T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, T17, T18, T19, T20, T21, T22, T23, T25, T26, T27, T28, T30, T31, T32, T33, T34, T35, T36, T37, T38, T39, T40, T41, T42, T43, T44, T45, T46, T47, T48, T49, T50, T51, T52, T53, T54, T55, T57, T61, T65, T66, T68, T70, T71

that they primarily focus on non-functional testing (e.g., T29, T59, T62, T63, and T69), such as performance testing. On the other hand, other tools offer support for functional testing (e.g., T24, T44, and T60), including regression testing. However, they usually require to manually write test cases. For instance, `DataDog` and `Sematext`, provide the possibility to define synthetic tests. When these tests are executed, user interactions are simulated by performing synthetic requests and actions on the applications' endpoints. Integration with testing is a possible future challenge, a deeper discussion follows in the next section.

8. Discussion

In this section, we put in context the results emerging from our analysis by presenting (i) the main findings and guidance for DevOps engineers (Section 8.1), (ii) open challenges to be addressed by both researchers and tool vendors (Section 8.2), and (iii) cross-cutting findings emerging from our orthogonal analysis (Section 8.3).

8.1. Main findings and guidance for DevOps engineers

Table 20 summarizes the main findings of our study, grouped by research question, together with high-level observations for helping DevOps engineers in selecting a monitoring tool for their own microservice-based system. The details of the specific features of each tool are provided in the referred sections in this article and in the replication package.

8.1.1. The ecosystem

The main takeaway message of this study is that the ecosystem of monitoring tools for microservice-based systems in the context of DevOps is **extremely active**, as there has been an important effort to develop and maintain tools in the last ten years. In this effort, web services and Microservices contributed to raising the market of monitoring tools initially devoted to distributed systems in general, favoring an increase in the number of tools (since around 2009 with a peak in 2014) tailored for service-based software. Nevertheless, such an ecosystem is also extremely **fragmented**: each of the analyzed monitoring tools has its own characteristics and can fit DevOps engineers with very different needs, *e.g.,* in terms of targeted quality attributes (*e.g.,* performance, security), monitored metrics (*e.g.,* CPU usage, network latency, energy consumption), applied monitoring patterns (*e.g.,* health check API, log aggregation), etc. So, it is more important than ever for DevOps engineers to make informed decisions when choosing the right tool for monitoring their own system.

We suggest DevOps engineers to **use our data extraction form (see** Table 3**) as a checklist** for guiding their decision process about the monitoring tool to use in their own system. Such a checklist can act as a compass when reasoning on the monitoring tool to use (and the implied trade-offs among the various dimensions of our data extraction form). To date, there is a vast choice for practitioners, with tools offering several features besides the basic data-collection facilities. While most tools offer some visualization and metrics reporting facility, only a few have advanced features, such as exposure of APIs for integration with other systems, customizable data searches, and analysis or optimization features. These are features that, if needed, will significantly restrict the space of possible choices, and they should be weighed against other characteristics, such as collected metrics, required technologies, and performance overhead. In this context, we suggest DevOps engineers use our data extraction form *incrementally*, *i.e.,* to consider the parameters based on the importance of the tool's features for the project at hand. Specifically, we identified three levels of parameters: the top level contains first-class features representing *tier-1 parameters* for the DevOps team, the second level is about *tier-2 parameters*, *i.e.,* those parameters whose values are still required by the DevOps team, but they are not blocking, and finally we have the third level containing *optional parameters*, which represent those parameters whose values are desiderata for the DevOps team. Based on the collected data and our experience in both industrial and academic projects, we propose the following concrete levels for choosing a monitoring tool (for each parameter we also provide concrete examples of questions DevOps engineers can ask themselves during the decision process):

- Tier-1 parameters:
  - Target – *What needs to be monitored, individual microservices, the system as a whole, etc.?*
  - Features/motivation – *Why does the DevOps team need to monitor the system (e.g., for visualization, reporting, optimization, etc..)?*
  - Assumptions – *Does the system/project satisfy all assumptions of the tool (a specific OS, Docker, specific DB technology)?*
  - Integration/Dependency tools – *Does the team have the capacity to bring up the technologies on which the tool depends (e.g., Apache Kafka)?*
  - Monitoring metrics (user-oriented) – *Which high-level metrics does the DevOps team need to collect from the running system (e.g., health, UX, user sessions)?*
  - Monitoring metrics (system-oriented) – *Which system-level metrics does the DevOps team need to collect from the running system (e.g., network requests, IO operations, CPU usage)?*

Table 20
Main findings of this study.

Main characteristics of the monitoring tools (RQ1)	Section
About half of the tools (34/71) have been **released** after 2014, in parallel with the boost of microservices and DevOps. Most of tools are actively **maintained** (77% of the tools have a last release in 2021 or 2022). Proprietary tools tend to have a longer lifetime.	4
MostL of the tools (58%) **target** high-level distributed systems, while the rest, especially newer ones, are specifically focused on web service technologies (21%) and for microservice-based systems (21%).	5.1
Most of the tools (48/71) offer their own **visualization and reporting** facilities besides monitoring; a significant share of the tools offer alerting (29/71) and data analysis (26/71) capabilities. More advanced features are available in a few tools, such as: custom data search/analysis (16/71), dedicated APIs for third-party components (10/71), and optimization features (4/71).	
The most widely supported **data export formats** are JSON (49/71) and CSV (25/71). Most of the tools (54/71) offer their own visualization features, such as charts, tables, and dashboards.	5.2
The most used **programming languages** for developing monitoring tools are: Go (27/71), Java (25/71), Python (22/71), and JavaScript (20/71); a non-negligible number of tools (27/71) use multiple languages.	5.3
The most used **technologies** are related to visualization (39%), data collection tools (35%), and databases (30%).	
Specific OS, technologies (e.g., Docker containers, JVM, Node.js, a SaaS), or libraries are explicitly stated as **requirements** in no more than 20% of the tools.	
The **challenges** (Waseem et al., 2021) addressed by more than 50% of the tools are those related to performance (MC4), metrics and logs (MC1), complexity (MC3), and data analysis (MC5).	5.4
Type of information collected by the tools (RQ2)	
The **user-oriented metrics** that are collected the most are custom/user-defined (26/71) or related to system failures (26/71), timing (24/71), and networking (23/71).	6
The **system-oriented metrics** that are collected the most are related to networking (55/71), memory usage (53/71), and CPU load (50/71).	
The majority of the tools (41/71) do not support **distributed tracing**, while 30/71 tools provide dedicated support for distributed tracing.	
The majority of the tools (54/71) support **failures/events logging**, while the remaining 17 tools do not support it.	
The most targeted **quality attributes** are performance (63/71), reliability (53/71), and security (15/71). Only 5 tools target user-defined quality attributes and only 3 tools target energy consumption.	
Realization of the monitoring infrastructure (RQ3)	
Almost all monitoring tools require an **instrumentation** step (68/71). Among them, the majority of the tools provide a vendor-specific platform (59/68), followed by a vendor-provided library (37/68) and either a third-party platform (36/71) or third-party library (36/71).	7
The most frequently-used **monitoring patterns** are: Application Metrics (44/71), Log Aggregation (39/71), and Health Check API (33/71). Out of the 71 monitoring tools, 11 tools use all 6 monitoring patterns we consider in this study.	
The vast majority of the considered tools have a **monitoring granularity** defined at the microservice level (62/71), followed by the VM/container level (42/71), infrastructure level (40/71), and finally application level (37/71).	
The majority of the analyzed tools do not have any **integration with testing activities** (60/71). The remaining 11 tools deal with testing in some way, e.g., by supporting canary releases, synthesis of test cases, or by providing dedicated testing environments for end-to-end testing of the system.	

  - Targeted quality attribute(s) – *Is the DevOps team interested on security-related aspects of the system? What about energy efficiency? What about performance?*

- Tier-2 parameters:
  - Open Source – *Does the DevOps team need to customize/modify the monitoring tool?*
  - Visualization – *How are the monitored metrics visualized? Is a graphical visualization needed? If yes, which one?*
  - Requests tracing – *Is it needed to collect information about all (internal) API calls made when executing a usage scenario?*
  - Events/Failures logging – *Does the DevOps team need precise information about specific events and failures within the system?*
  - Instrumentation – *Are there resources, skills, and time available for instrumenting the monitored services?*

- Optional parameters:
  - Provider – *Does the DevOps team already have a business relationship with the tool provider? Does the DevOps team need support from the tool provider?*
  - Available format(s) to export data – *Is it required to analyze the monitoring data externally? If yes, are JSON or CSV (or other) file formats acceptable?*
  - Addressed Challenges – *Are there any orthogonal relevant aspects about the tool and the system that should be taken into consideration*
  - Monitoring granularity – *Does the DevOps team need to monitor the application-level metrics, individual microservices, the infrastructure, etc.?*
  - Monitoring patterns – *Which monitoring patterns is the DevOps team familiar with (e.g., health check API, audit logging, exception tracking)?*
  - Monitoring practices – *Which monitoring practices is the DevOps team familiar with (e.g., deployment logging, log aggregation, etc.)?*
  - Programming language(s) – *Does the DevOps team need to customize the monitoring tool? If yes, which technologies/programming languages are they familiar with?*
  - Integration with testing – *Does the DevOps team have an already-in-place testing infrastructure that needs to be integrated with monitoring data?*

The above-mentioned levels can be used as follows. Tier-1 parameters are defined *a priori* by DevOps engineers and guide the first round of filtering of available monitoring tools; the line of reasoning is that tools passing this first filtering step provide a satisfactory coverage of *all* tier-1 parameters. This phase also helps DevOps engineers in prioritizing what is really important for their monitoring policies. Then, those monitoring tools that have not been filtered out will undergo a second filtering phase based on tier-2 parameters. In this phase, the selection is less stringent, a tool passing this phase might not provide some features for the selected tier-2 parameters (as a rule of thumb, we might expect that a selected tool might provide at least 80% of the required values for tier-2 parameters). Finally, the remaining tools undergo a third selection phase, where the tool providing the majority of the optional features is finally selected and used in the project. The selection based on tier-2 and optional parameters is iterative and incremental, meaning that also the requirements for the monitoring tool can be refined and re-prioritized during the selection itself. It is important to note that the levels proposed above are our attempt to extract the dimensions that can possibly fit most software projects. We invite practitioners to carefully assess if our proposed levels fit their project and, in case they do not, to adapt them according to the project's requirements and technological/organizational context.

8.1.2. First of all, why monitoring

More in general, we advise DevOps engineers to reason in a top-down fashion when deciding which monitoring tool to use, starting with the **why monitoring is done** from an organizational point of view. Examples of questions to be asked here include: *is monitoring done for receiving real-time alerts about system malfunctions (i.e., reliability engineering)? Is monitoring done mostly for collecting logs to be used in an external audit (this scenario is specifically useful for highly-regulated domains like finance)? Is a real-time dashboard showing the collected metrics needed/used? If yes, who is using it (e.g., DevOps engineers, business analysts, customers)?* The main dimensions to be considered when taking these decisions include: tools' features/motivation, targeted quality attributes, user-oriented and system-oriented metrics, and data visualization means.

8.1.3. Required technologies and assumptions

The choice of a tool is also heavily related to the required **technologies for the tool to run or work properly**. These latter ones are most often related to visualization (39%), data collection tools (35%), and DB (30%). But other categories might also be relevant; for instance, some tools require virtualization to work or have requirements on configuration management. This needs to be read together with the list of **assumptions**, as these two dimensions can significantly restrict the possible choices, depending on the user's needs. In this regard, we found that the documentation of the tools reports the specific operating system required, the libraries needed, or the required technologies (e.g., containers, JVM, Node.js), or reports about requirements for the system under monitoring (e.g., only Java-based applications are monitored). This is however explicitly reported in no more than 20% of the tools, which does not mean that the rest of the tools is free from assumptions — we advise the reader to check this aspect case-by-case.

8.1.4. Distributed tracing

In our dataset, 30 monitoring tools support distributed tracing. This is not a surprise per se since using distributed tracing tools of microservice-based systems is becoming the state of the art, specially in the context of anomaly detection and performance analysis (Bento et al., 2021; Huye et al., 2023). However, distributed tracing tends to be more complex and resource-demanding than the collection of individual metrics for each service (Shkuro, 2019); this means that potentially distributed tracing might lead to a higher overhead for the system being monitored (Bento et al., 2021). We suggest DevOps engineers to critically reflect on whether the usage of distributed tracing will pay off for them in terms of, e.g., higher system observability and early diagnosis in case of failures or performance regressions. We also suggest DevOps engineers to experiment with different configurations of the tracing tool (e.g., about the sampling frequency of the traces) in order to ensure that the added overhead due to distributed tracing is still bearable for the system as a whole.

8.1.5. Instrumentation

As shown in Table 20, almost all monitoring tools require instrumentation. This means that the source code of the microservices being monitored must be extended or annotated with probes that suitably collect the metrics, logs, and traces of interest. Instrumentation code might be relatively simple (e.g., a basic probe) or more complex (e.g., for creating a span in a distributed tracing tool and assigning it to the correct trace id); in any case, it is additional code that is developed, maintained, and operated by (potentially different) development teams. We advise DevOps engineers to (i) **choose the monitoring tool whose instrumentation fits well** with the development pace of the system (some of them, like `Jaeger`, `Prometheus`, `Zipkin`, and `elasticsearch` support some level of automatic instrumentation via `OpenTelemetry` libraries.[8]) and (ii) allocate proper time and resources towards the **co-evolution of the microservices source code and their instrumentation code**

8.1.6. Community support

Finally, also the state of the community around the chosen monitoring tool plays a strong role. Some of the analyzed tools have a lively open-source community (e.g., Prometheus, with its 49+ thousand stars on GitHub and well-defined contribution strategy), making them good candidates in terms of long-term support. Some of the open-source tools are also backed by nonprofit foundations such as the Apache Software Foundation (`Apache skywalking` – T3) or the Cloud Native Computing Foundation (`Prometheus` – T1 and `Jaeger` – T4), thus guaranteeing a certain level of transparency and support over the years. Other open-source tools are instead maintained by companies such as Amazon (`AWS CloudWatch` – T29) or Elastic (`elasticsearch` – T23). Differently, other tools are either closed-source or maintained by a single contributor, resulting in a riskier investment for DevOps engineers.

8.2. Open challenges for researchers and tool vendors

8.2.1. Open challenges for researchers

Researchers can use our classification of the 71 monitoring tools to get a detailed overview of the characteristics of existing monitoring techniques and use it to either (i) **steer their own research** towards methods and techniques that are still not covered by existing tools or (ii) **identify monitoring tools which can be reused** as building blocks in their own research on DevOps and microservices. Below we report about the promising research gaps we noticed while analyzing the collected monitoring tools:

- Assessment of **runtime overhead** of monitoring microservice-based systems: the documentation of several monitoring tools claims that the tool is highly efficient and with low overhead in terms of usage of resources (e.g., CPU, memory, networking, energy). However, to the best of our knowledge there is no scientific study providing empirical evidence about such overhead. Also, an independent assessment carried out by researchers (not affiliated with any organization behind the tools) will provide objective, trustable, and replicable insights about this particular aspect of monitoring tools for microservices.
- **Instrumentation bugs**: as mentioned in the previous section, the instrumentation code in a microservice is still code developed by the team responsible for the microservice. As such, the risk of introducing bugs in the instrumentation code is there and (to the best of our knowledge) it has not been studied yet. In this context it might be interesting to (i) characterize instrumentation bugs (e.g., via a study mining software repositories), (ii) assess the possible consequences of those bugs in terms of the correctness of the produced metrics and traces, and their potential impact on the decision process of DevOps engineers, and (iii) propose (semi-) automated approaches for detecting and solving instrumentation bugs.
- **Impact of misconfiguration of monitoring tools**: this research line is somehow in between the previous two (if we consider a misconfiguration as a form of bug), but it is different. Monitoring tools can be configured in several ways, As an example, the majority of the tools supported distributed tracing can sample the collected traces at different frequencies, in an adaptive manner,

[8] https://opentelemetry.io/ecosystem/integrations

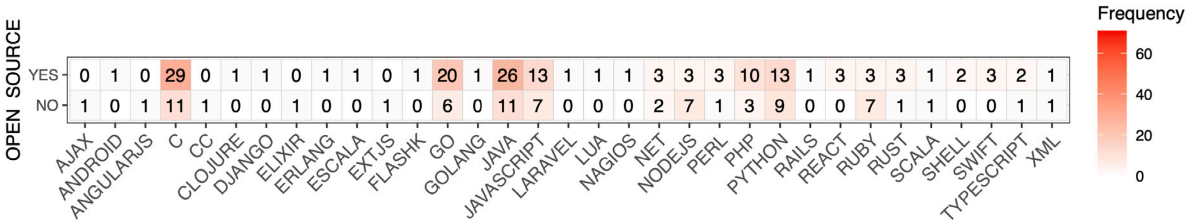


Fig. 7. Co-occurrences between *open§/closed source* and *programming language*.

based on rules, etc. All of those configurations might potentially lead to issues with respect to the correctness of the produced metrics or unexpected runtime overhead. It might be interesting to characterize, assess, and measure how monitoring tools behave under different configurations and on their impact on the overall quality of the system being monitored.

8.2.2. Open challenges for tool vendors

Tool maintainers can use our map of 71 monitoring tools to identify competing tools and avoiding to reinvent the wheel. We also identified potentially-interesting gaps within the monitoring tools landscape that tool vendors can use to anticipate the features of their next generation monitoring tools. Below we report about those identified gaps:

- **Integration with testing activities**: testing and online experimentation via A/B testing procedures and canary releases are the norm today when dealing with Cloud-based applications, so it strikes the eye that the majority of the analyzed tools do not have any integration with testing activities. Some tools have it, but they are a minority with respect to the main trend. Tool vendors are invited to explore further how testing how can be integrated in their monitoring tools.
- **Target unaddressed challenges of microservice practitioners**: the Failure zone detection (MC6), the Monitoring of applications running inside containers (MC8), and Maintaining monitoring infrastructures (MC9) are the least-addressed challenges in our extracted data. Features addressing those challenges are intrinsically promising for future releases of monitoring tools since they will be addressing concrete issues and concerns voiced by microservice practitioners, as emerged in Waseem et al. (2021).
- **Monitor power consumption**: only four tools monitor the power consumption of the nodes where the microservices are running (T1, T6, T10, and T59). This is a missed opportunity since the energy demand of microservice-based systems is exploding (Verdecchia et al., 2021) and society and policy makers are starting to build a sensibility on the energy consumption of Cloud-based software services in general. Interestingly, none of them are providing the power consumption at the single-microservice level. This might be an opportunity for tool vendors since in their tools they might unlock further features, such as (i) the identification of energy hotspots in the monitored system (i.e., those services that are particularly energy-hungry), (ii) the support for root cause analysis in terms of power consumption, and (iii) the support for microservices redeployment based on their current power consumption and/or temperature of the processor where they are running.
- **Better integration with maintainability**: Only one tool (i.e., T1) targets the maintainability of the system. This is also a missed opportunity since it might be informative for DevOps engineers to have an integrated view of the development activities and the runtime metrics of each monitored microservice. For example, we might think about having a dashboard showing information about pushes on the GitHub repository containing the source code of a microservice, its CD/CI actions (e.g., automated builds and deploys), and variations of its runtime metrics like CPU and memory usage; with such an instrument DevOps engineers might easily spot performance regressions in their managed microservices, without needing to move from one tool to another risking to lose precious contextual information.

8.3. Cross-cutting findings

This section describes the results of our orthogonal analysis. The goal of the orthogonal analysis is to investigate possible co-occurrences between related dimensions of the classification scheme (see Section 3.3). Specifically, firstly we collaboratively identified 21 pairs of dimensions whose co-occurrences can lead to potentially-interesting cross-cutting findings, then we built contingency tables for the identified pairs of dimensions, we analyzed each one of them, and finally synthesized the most interesting cross-cutting findings emerging from our analysis. In the remainder of this section, we present the cross-cutting findings emerging from 7 of the initial 21 pairs, one pair in each subsection . We do not report the results of the other 14 pairs since they either did not exhibit observable trends or did not lead to additional insights with respect to those of the vertical analysis.[9]

8.3.1. Open source and programming languages

Fig. 7 reports the co-occurrences between open/closed source and programming language. In line with the results of our vertical analysis, **C (29 open-source vs 11 closed-source)**, **Java (26 open-source vs 11 closed-source)**, and **Go (20 open-source vs 6 closed-source)** are the most used programming languages in open-source projects. We suggest to junior practitioners who want to enter the open-source monitoring tools ecosystem to specialize in at least one of the three above-mentioned programming languages. There is slightly more balance when considering **Python**-based projects **(12 open-source vs 9 closed-source)** and even an opposite trend when considering **Node** projects **(3 open-source vs 7 closed-source)**. The latter result is interesting, especially due to the popularity of the Python and Javascript languages today (TIOBE, 2022). The most recently-created monitoring tools in our dataset are both open-source and developed in Java. Those tools are `easeagent` (T55) and `OpenSignals` (T57) and both of them are dedicated to monitor Java-based microservice-based systems.

8.3.2. Open source and addressed challenges

We crosschecked the challenges addressed by each monitoring tool and whether it is an open-source project or not. Fig. 8 reports the co-occurrences. In this way, we can assess the open-source community and evaluate how open-source tools are able to help practitioners in addressing their challenges. In line with the results of the vertical analysis, the majority of identified challenges are targeted more by **open-source tools** (which are **65%** of all analyzed tools in total) rather than **closed-source** ones (which are only **35%** of all analyzed tools in total). Nevertheless, we identified an opposite trend when looking at the *maintenance of the monitoring infrastructure challenge* (MC9). Indeed,

[9] For transparency, all contingency tables and our extracted findings are available in the replication package, allowing for further analysis by the interested reader.

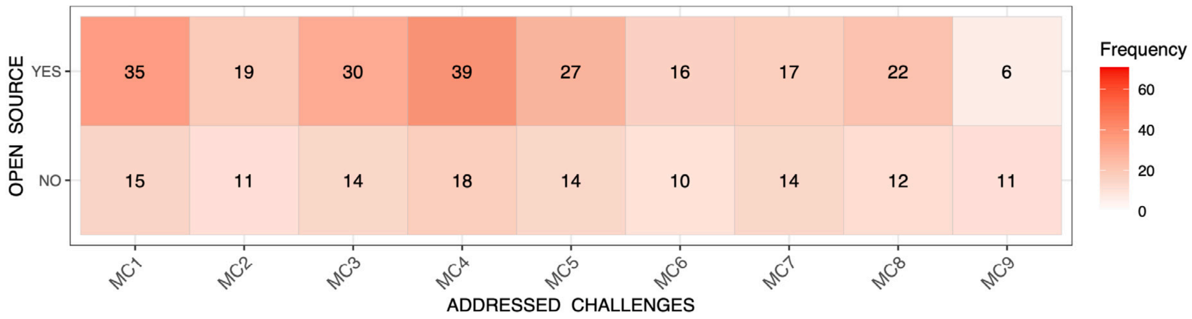


Fig. 8. Co-occurrences between *open§/closed source* and *addressed challenges.*

only **6 open-source tools target MC9**, as opposed to **11 closed-source tools** targeting it. The 6 open-source tools targeting MC9 are: `elasticsearch` (T23), `AWS CloudWatch` (T29), `Sensu` (T31), `netdata` (T40), `vigil` (T53), `Reimann` (T64). We invite the open-source community to fill this gap by providing more support for the maintenance of the monitoring infrastructures operated via their open-source monitoring tools. Possible action points to address this issue include (but are not limited to): (i) *providing better support in terms of co-evolution of the monitored services and the sidecar services/agents monitoring them, (ii) supporting standard formats for representing monitored data*, such as `OpenTelemetry`, and *(iii) better support the migration towards newer releases of the monitoring tool*, without requiring a reboot of either the monitoring tool or the monitored services, etc.

8.3.3. User-oriented metrics and system-oriented metrics

Fig. 9 reports the co-occurrences between user- and system-oriented metrics. It does not come as a surprise that the most frequently-used user-level metrics (e.g., those about timing, networking, and failure) co-occur with the most frequently-used system-level metrics (e.g., those about CPU load, IO operations, memory usage, network traffic, DB usage). In this case we did not observe any significant gap to be filled by researchers and tool vendors. However, we noticed interesting results when analyzing the co-occurrences of user-oriented metrics (see Section 8.3.4) and system-oriented metrics (see Section 8.3.5).

8.3.4. Co-occurrences of user-oriented metrics

Fig. 10 reports the co-occurrences between different user-oriented metrics. The user-level metrics that co-occur more frequently in our collected data are the following: **Timing and Failure metrics (19 co-occurrences)**, **Timing and Networking metrics (18 co-occurrences)**, and **Failure and Networking metrics (17 co-occurrences)**. Those co-occurrences are expected since Timing metrics (e.g., system overall latency, average response time) can strongly depend on possible system failures, and availability, and its communication infrastructure. **Custom metrics** defined by DevOps engineers also tend to co-occur with **timing metrics (13 co-occurrences)**; we speculate that the latter is an indication of the fact that raw timing metrics might not always be enough to observe the overall system health and monitoring tools provide means for allowing DevOps engineers to add their own custom metrics, such as the well-known “Time to First Tweet”, defined as: “the amount of time it takes from navigation (clicking the link) to viewing the first Tweet on each page’s timeline” (Firtman, 2018). In a recent industrial case study we empirically observed that product-specific metrics like the “Time to First Tweet” exhibit a perfect correlation with the user-perceived load time, thus proving higher value with respect to generic/raw performance metric (Riet et al., 2023). We also observed two interesting gaps: *current monitoring tools never support at the same time user-oriented metrics covering (i) Security and DBs metrics and (ii) Container lifecycle and Failure metrics*. Those two gaps might be opportunities for tool vendors willing to expand the features of their tools in terms of observability capabilities at a higher level of abstraction than that of system level.

8.3.5. Co-occurrences of system-oriented metrics

Fig. 11 reports the co-occurrences of different system-oriented metrics. As expected, the most frequent co-occurrences are about system metrics that are frequently used when monitoring Cloud-based systems, such as: **network traffic and memory usage (49 co-occurrences)**, **network traffic and CPU load (44 co-occurrences)**, **network traffic and I/O operations (41 co-occurrences)**, **I/O operations and memory usage (44 co-occurrences)**, **I/O operations and CPU load (42 co-occurrences)**. We observed potentially-interesting gaps related to the *power consumption* of the system. Indeed, even though energy and power consumption are being monitored by multiple Cloud vendors (Verdecchia et al., 2021), the monitoring tools providing power consumption metrics (i.e., T1, T6, T10, T59) do not provide other metrics that are conceptually strongly linked to power consumption. Specifically, according to our analysis, there is *no monitoring tool that supports at the same time metrics about power consumption and (i) the temperature of the processors, (ii) system load, (iii) container lifecycle, and (iv) system failures*. We invite vendors of monitoring tools to support the four previously-mentioned metrics since they can help DevOps engineers in (i) better understanding the reasons they might observe peaks of power consumption in their system and (ii) finding solutions for reducing the overall power consumption of their systems. As an example of such solutions that can be achieved when using power metrics combined with other ones, we mention `Kube Green`.[10] `Kube Green` is a Kubernetes addon that automatically shuts down pods in Kubernetes-based systems when they are not strictly needed (i.e., dev/testing pods outside office hours); in this case, the status the lifecycle of each container might be monitored in combination with the power consumption of the system in order to semi-automatically trigger Kube Green and turn off selected containers based on their lifecycle status.

8.3.6. Requests tracing and targeted quality attributes

Fig. 12 reports the co-occurrences of the *Tracing* parameter (i.e., the tool supports request tracing) with quality attributes. Being performance and reliability the most targeted quality attributes in our dataset, they are also the ones with higher co-occurrences with the *Tracing* parameter. Here we can see a certain balance for **performance**, where **29** monitoring tools **provide support for distributed tracing**, as opposed to **34** monitoring tools **not supporting it**. An example of tools supporting distributed tracing and targeting performance is `Zipkin` (T2), which allows DevOps engineers to diagnose latency problems by collecting traces of service calls annotated with timing information. The data about **reliability** is relatively similar, but less balanced, with **21 tools supporting distributed tracing versus 32 tools not supporting it**. An example of tools supporting distributed tracing and targeting reliability is `Jaeger` (T4), which includes in its produced traces also error codes of the requests made within each trace, in addition to timing information (thus covering also performance). We speculate that for

[10] https://github.com/kube-green/kube-green

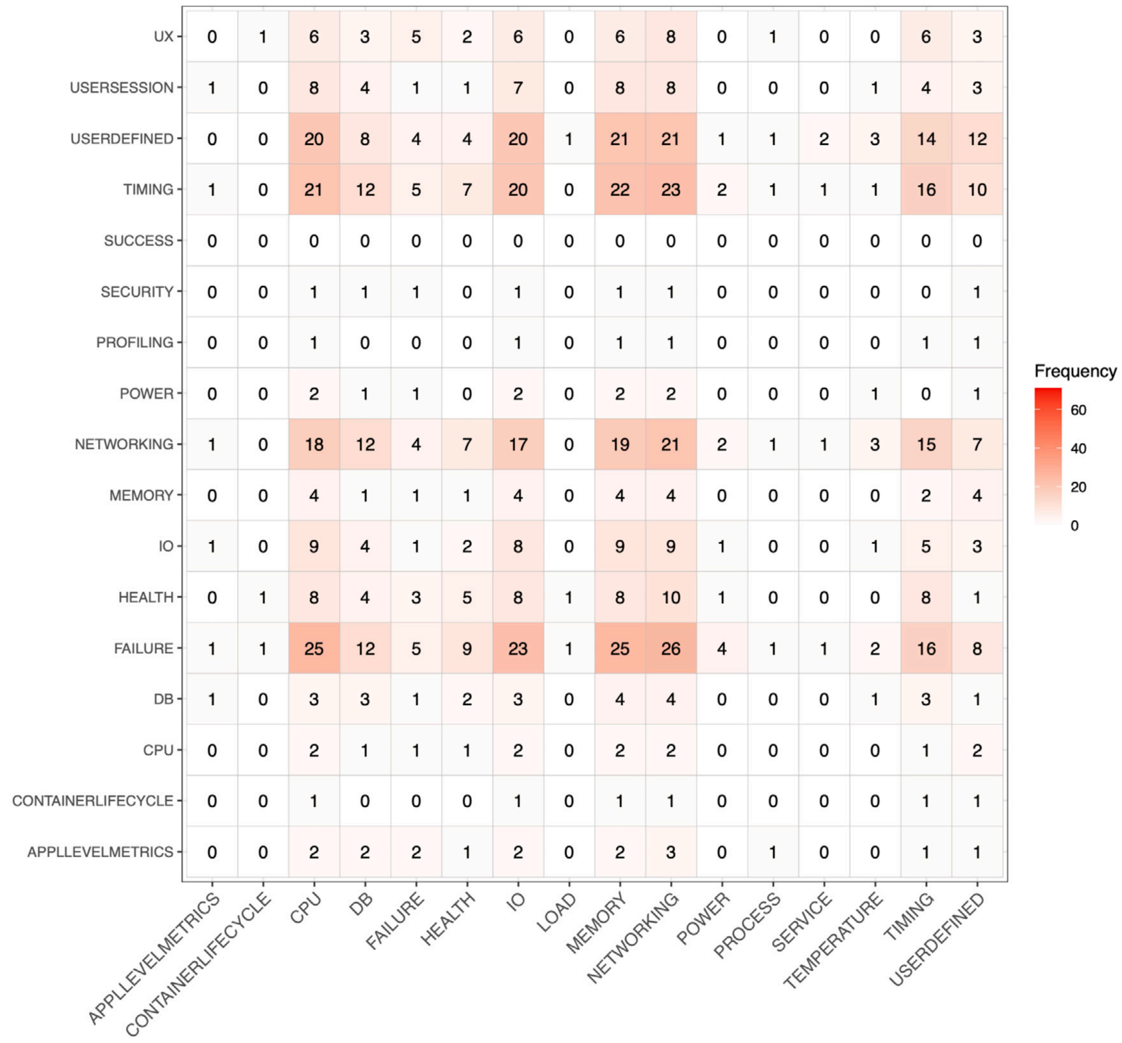


Fig. 9. Co-occurrences between *user-oriented* and *system-oriented metrics*.

DevOps engineers the choice of having a monitoring tool supporting distributed tracing boils down to organizational constraints related to the required level of observability of the system; this decision is important since distributed tracing does not come for free, tracing can add significant overhead to the system (thus impact performance), several tools supporting tracing require instrumenting the services being monitored, analyzing traces might be non trivial, specially with systems with failover mechanisms (thus leading to different paths under the same scenarios), etc. When taking this decision, examples of questions that DevOps engineers might ask themselves include: *do they need to understand the behavior of the system as a whole? Will chains of service calls be audited either internally or by an external body in the future? Etc.*

8.3.7. Testing and targeted quality attributes

Fig. 13 reports the co-occurrences of the parameter *Testing* (hence the tool is integrated with testing) and quality attributes. **All monitoring tools targeting the performance** of the system also provide some **integration with testing**; similarly, **9** out of the 11 tools supporting testing **target reliability**. This result is not surprising due to the fact that performance and reliability are by far the most targeted quality attributes in our dataset. `Amazon CloudWatch` (T29) is an example of monitoring tool supporting testing and targeting both performance and reliability. `Amazon CloudWatch` provides the concept of canary, which is a script written either in Node.js or Python implementing an end-to-end test case. While executing canaries `Amazon CloudWatch` can collect timing metrics (e.g., loading time of a web page), the number and type of successful and failing HTTP(S) requests together with their response codes, and screenshots of the UI. The execution of canaries can be triggered either manually by DevOps engineers or on a schedule. However, security aspects are targeted by only 3 monitoring tools, i.e., `Splunk` (T60), `DataDog` (T67), and `Akamai mPulse` (T69), and none of them is open-source. As an example, `Splunk` allows DevOps engineers to (i) setup a small-scale testing environment mirroring the topology of the system in production, (ii) define realistic test data, and (iii) generate test cases for specific aspects of their `Splunk` extensions. Interestingly, we did not observe any monitoring tool integrated with testing that targets either **compatibility** or **energy**. These might be two promising research directions for the software engineering community. Recently, some steps are being done on green testing of Cloud applications (Verdecchia et al., 2021), i.e., the practice of assessing the energy consumed by running test cases. This problem is attracting the attention of researchers since it has been observed that many teams produce, maintain, and run test cases without any strong underlying test strategy, wasting resources (and producing carbon emissions).

9. Threats to validity

The study is subject to the following threats to validity.

Data collection and analysis. There are factors that can affect the initial phases of data collection and the subsequent analysis:

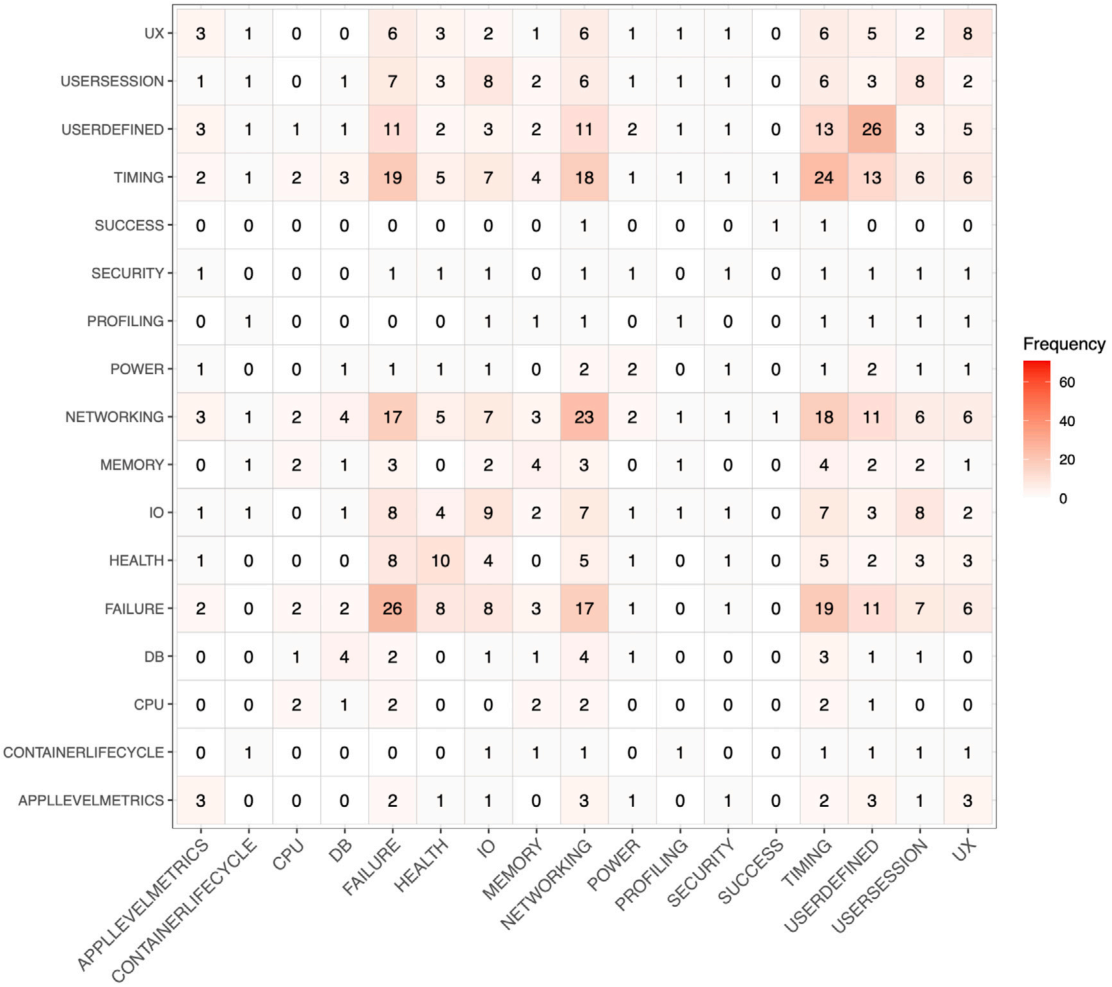
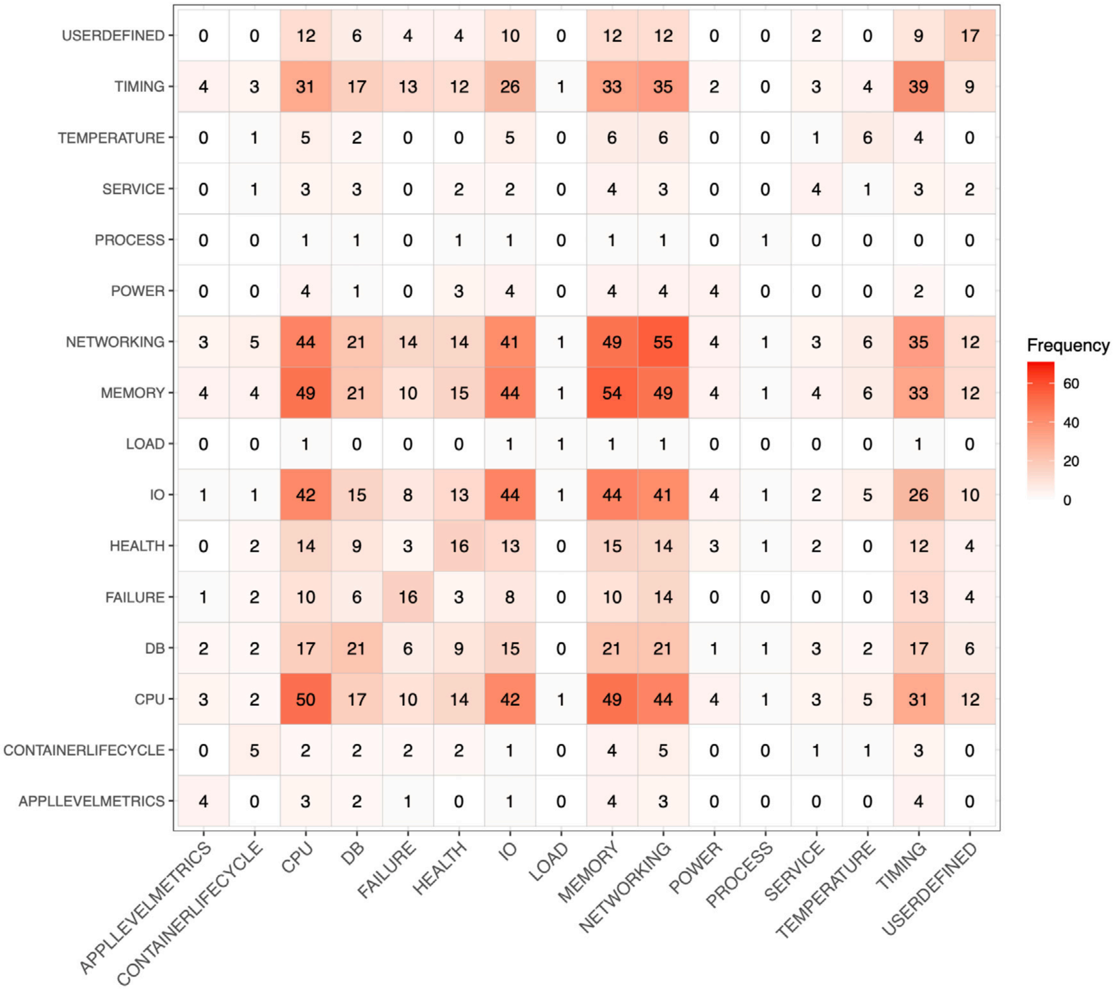


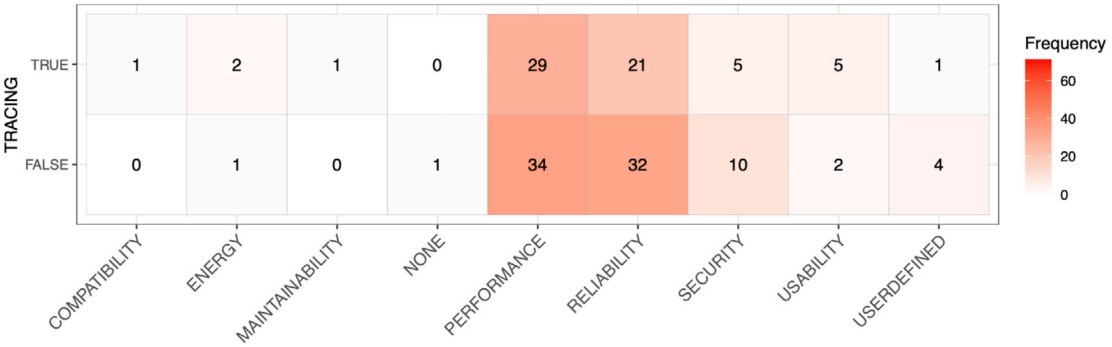
Fig. 10. Co-occurrences between different *user-oriented* metrics.

- *Tools selection*. The study is conducted on a set of 71 monitoring tools (from 181 in the initial list), which could be a non-representative set, as some relevant tools might have be missed. To reduce this risk, the protocol we have adopted foresees (i) a selection from the most known source for software repositories (with 372 million repositories) complemented via Google search, and with a search string kept very general to be conservative; (ii) a two-step procedure including selection of sources (*e.g.,* repositories, web pages, which may contain multiple tools) followed by selection of tools; (iii) the reference test set strategy (Petersen et al., 2015; Kitchenham and Brereton, 2013), to double-checked the quality of the selection with respect to widely-known tools; (iv) we double-checked the list with the industrial partners of the uDEVOPS project funding this work, to confirm that industry-relevant tools they know were in the list; (v) a set of documented inclusion and exclusion criteria was used to refine the search unambiguously.
Despite the adopted protocol, some relevant tools might have been missed, *e.g.,* excluded because not accessible, or because not focused on monitoring as primary task (but that can include some monitoring facilities), or because not present either in the top 100 results by Google or in the GitHub repository. Overall, we think the set of actions taken and the protocol followed reduce this possibility; in particular, searching from two top sources (GitHub and Google), and double checking both against a test set and with industrial partners, go beyond the best practices suggested for literature reviews (Petersen et al., 2015; Kitchenham and Brereton, 2013). With these actions, the absence of some relevant tool is not expected to significantly alter the discussed findings and provided suggestions, stemming from a set of 71 tools.
- *Data extraction and classification*. Data extraction and classification could also be biased by a subjective interpretation. The analysis was carried out by four research teams on a scheme of 26 dimensions aimed to cover a wide range of aspects. To reduce the effect of subjective interpretation by the teams, we have (i) devoted the initial effort to agree on the meaning of dimensions and terms used, by classifying a same subset of tools and discussing the results in plenary meeting; (ii) adopted an iterative approach, where we get to the final classification scheme by successive refinements coming from the analyzed tools, to guarantee that the data extraction process was aligned with the research questions; (iii) we involved the industrial partners to be sure that the identified dimensions (*e.g.,* quality attributes monitored, monitoring techniques, challenges) were relevant from a practitioners perspective; (iv) we actually have used some of these tools (`Prometheus`, `Zipkin`, `ELK Stack`, `NetData`, `Jaeger`) in the development process of one the industrial partners to analyze their pros and cons, which allowed us to infer features of interest and refine our scheme. One of the dimensions referring to "challenges" are taken from previous works (Waseem et al., 2021), which drawn them from surveys with practitioners; indeed, there might be other interesting challenges for practitioners that we missed.



	APPLLEVELMETRICS	CONTAINERLIFECYCLE	CPU	DB	FAILURE	HEALTH	IO	LOAD	MEMORY	NETWORKING	POWER	PROCESS	SERVICE	TEMPERATURE	TIMING	USERDEFINED
USERDEFINED	0	0	12	6	4	4	10	0	12	12	0	0	2	0	9	17
TIMING	4	3	31	17	13	12	26	1	33	35	2	0	3	4	39	9
TEMPERATURE	0	1	5	2	0	0	5	0	6	6	0	0	1	6	4	0
SERVICE	0	1	3	3	0	2	2	0	4	3	0	0	4	1	3	2
PROCESS	0	0	1	1	0	1	1	0	1	1	0	1	0	0	0	0
POWER	0	0	4	1	0	3	4	0	4	4	4	0	0	0	2	0
NETWORKING	3	5	44	21	14	14	41	1	49	55	4	1	3	6	35	12
MEMORY	4	4	49	21	10	15	44	1	54	49	4	1	4	6	33	12
LOAD	0	0	1	0	0	0	1	1	1	1	0	0	0	0	1	0
IO	1	1	42	15	8	13	44	1	44	41	4	1	2	5	26	10
HEALTH	0	2	14	9	3	16	13	0	15	14	3	1	2	0	12	4
FAILURE	1	2	10	6	16	3	8	0	10	14	0	0	0	0	13	4
DB	2	2	17	21	6	9	15	0	21	21	1	1	3	2	17	6
CPU	3	2	50	17	10	14	42	1	49	44	4	1	3	5	31	12
CONTAINERLIFECYCLE	0	5	2	2	2	2	1	0	4	5	0	0	1	1	3	0
APPLLEVELMETRICS	4	0	3	2	1	0	1	0	4	3	0	0	0	0	4	0

Fig. 11. Co-occurrences between different *system-oriented* metrics.



TRACING	COMPATIBILITY	ENERGY	MAINTAINABILITY	NONE	PERFORMANCE	RELIABILITY	SECURITY	USABILITY	USERDEFINED
TRUE	1	2	1	0	29	21	5	5	1
FALSE	0	1	0	1	34	32	10	2	4

Fig. 12. Co-occurrences between the *support for tracing* and *quality attributes*.

- *Quality*. Besides the relevance of selected tools discussed above, which clearly is also related to the quality of the provided artifacts, we have worked to define unambiguous inclusion and exclusion criteria to distinguish mature tools from just temporary code (such as a repository containing proof-of-concept or very rough prototypes not meant to become a tool). The accuracy of our analysis is also tied to the quality and accuracy of the tools documentation. This threat is slightly mitigated by the selection and analysis process, which involves independent classifications by the teams and plenary meetings to discuss ambiguities. To further reduce it, we published the final categorization on the online repository,[11] and as Zenodo resource (uDEVOPS2020, 2023), as supplemental material of this manuscript, thus making our analysis easy to be replicated by other researchers.
- The final findings are derived with reference to microservices and DevOps, although several tools are for distributed systems in general. Some considerations could change if applied to other distributed system domains (*e.g.,* the challenges in other cases

[11] https://github.com/uDEVOPS2020/Monitoring-Tools-for-DevOps-and-Microservices-a-Systematic-Study

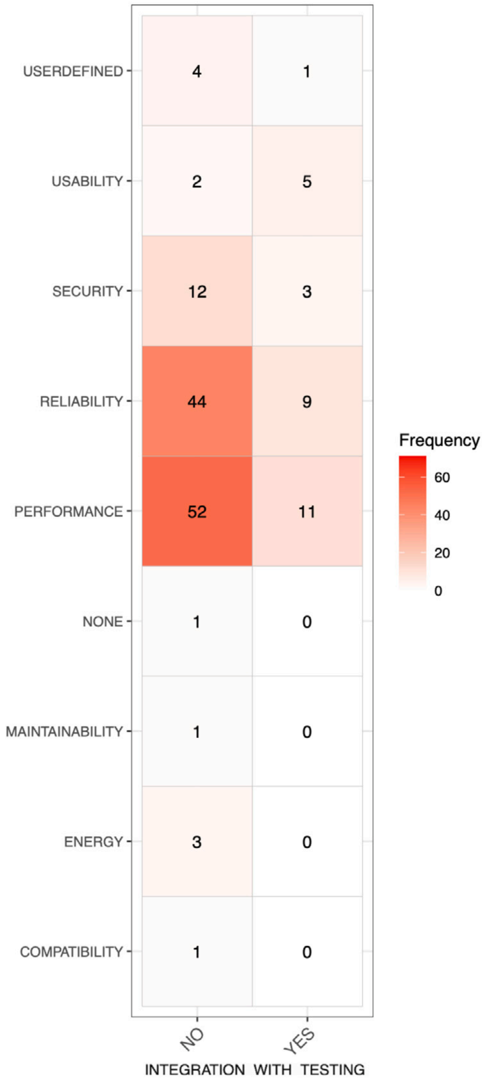


Fig. 13. Co-occurrences between the *integration with testing* and *quality attributes*.

could be different); therefore care must be taken in transferring the findings to domains outside microservices and DevOps.

In general, we tailored the best practices for mapping studies (Petersen et al., 2015) and literature reviews (Kitchenham and Brereton, 2013) for defining our protocol for tools analysis. This helped us to reduce ambiguities and improve the generalization and usefulness of our findings and suggestions.

10. Related work

In this Section, we report about surveys and secondary studies conducted on microservices, and their relation with our study.

10.1. Survey studies

In recent years, researchers have conducted survey studies concerning several aspects of microservice-based systems. Waseem et al. conducted a comprehensive survey with 106 participants and 6 interviews with practitioners (Waseem et al., 2021), investigating aspects ranging from design to monitoring and testing of microservices. They identified a list of 9 challenges that we have exploited in this study to check to what extent the existing tools tackle them. The survey includes a list of 13 monitoring tools the authors used in their survey.

Knoche et al. conducted a survey with 71 participants, exploring the challenges faced upon the need of modernizing legacy systems. Their analysis includes the impact of using microservices on runtime performance (Knoche and Hasselbring, 2019), which is clearly affected by monitoring too.

Viggiato et al. surveyed the practices adopted in industry for development and use of microservices, such as the adopted programming languages and technologies, as well as the advantages and challenges brought by microservices (Viggiato et al., 2018). The survey involved 122 participants. Among the challenges, they point out the microservices testing, faults diagnosing and distributed transactions.

Challenges related to the distributed nature of microservice architectures, among others, are also inferred by Ghofrani et al. in a survey study with 25 practitioners (Ghofrani and Lübke, 2018). These are clearly related to the challenges of monitoring we considered in this paper (identified in Waseem et al. (2021)).

Another survey with 21 participants was conducted by Wang et al. (2021), about development of microservices, at architectural, infrastructural and code management level. They identified the investment in robust logging and *monitoring* infrastructure, among others, as a best practices for successfully developing microservices.

There are some other surveys with similar analyses, but with less than 20 participants, hence with a more limited external validity, such as Haselböck et al. (2018), Zhou et al. (2021), Zhang et al. (2019).

These works focus on conducting surveys with practitioners, and their outputs are valuable for identifying main needs and challenges. Our study can complement these results by investigating whether and how existing tools in the grey literature fulfill these needs and address specific challenges.

10.2. Systematic literature reviews

Besides the surveys, researchers have conducted mapping studies and systematic literature reviews on microservices and DevOps. Di Francesco et al. looked at the state of the art on architecting activities with microservices (Di Francesco et al., 2017), defining a classification framework for categorizing the research on architecting microservices including architectural solutions, methods, and techniques (e.g., tactics, patterns, styles, views, models, reference architectures, or architectural languages). This was applied a set of 71 selected studies.

The same team later extended the work, by including further primary studies (from 71 to 103), and elaborating more on extracted data looking at the interactions between various parameters of the classification framework (Di Francesco et al., 2019).

Soldani et al. reviews the grey literature to depict an overview about the academic research and industry practices starting from 51 selected industrial studies, focusing on the technical/operational “pains” and “gains” of micro services (Soldani et al., 2018). They taxonomically classify, and systematically compare the pains and gains of micro services from existing grey literature, from design and development point of view. The challenges they identified (i.e., the “pains”) pertain (at development time) to the management of distributed storage and application testing, and (at operational time) to large consumption of network and computing resources compared to other architectural styles. Part of this consumption is indeed due to the features of the monitoring tools and their configuration, that we discussed in this paper.

Another work by Waseem (Waseem et al., 2020), that precedes the above-discussed survey, focus on identifying and classifying the literature on micro services in DevOps, starting from a set of 47 primary studies. This interestingly includes DevOps explicitly in the study. Their extensive analysis outlines the state of the art (at 2018) about all the phases of development and operation (requirements, design, implementation, testing, deployment, monitoring, organization, resource management), including the analysis of MSA description methods, patterns, qualities attributes, support tools, application domains, and research opportunities. The work identified a set of 11 monitoring tools, which are however not analyzed as it was only one of the many aspects covered by the paper and was not the focus of their paper. Our study, in contrast, is entirely focused on monitoring tools; we have extensively identified and analyzed available tools in the grey literature and provided recommendations on their selection and usage.

Although relevant and very interesting, none of these studies conduct a systematic review and comparison of *monitoring tools*. Several of them identified challenges related to monitoring, whose impact is deemed of extreme importance by practitioners. Indeed, having a stable monitoring infrastructure is a strong requirement for operating micro service-based systems where relevant incidents (e.g. faults, performance issues, security breaches) are promptly detected and diagnosed, and our aim was to provide a comprehensive view about the pros and cons of the existing tools. Our findings are therefore complementary, and to some extent orthogonal, to all the previous papers.

11. Conclusion

Monitoring is a key phase of DevOps and a key enabler for microservice-based systems. Because of the frequent releases, the highly changing nature of these systems and of their runtime context, monitoring is of paramount importance to continuously drive development and testing, starting from the feedback collected from the field. Key decisions about effort allocation for design and testing, architectural alternatives, deployment decisions, fault tolerance, diagnosis and fault removal, fault forecasting actions are enabled by monitoring. Consequently, the careful selection of a tool suited for specific needs can mark the difference in service-based software development and operation. It is noteworthy that monitoring tools frequently incorporate components or operate in conjunction with other tools to carry out more complex tasks. An example is AIOps (Artificial Intelligence for IT Operations), where tools leverage advanced analytics, machine learning, and automation to enhance and optimize IT operations. As part of our future work, we intend to expand this study with a detailed analysis of tool integrations and their interactions.

With this study, our aim was to give a systematic overview of the available choices for monitoring tools for microservices and DevOps, as well as to analyze the ongoing efforts, the challenges and directions that are being pursued to offer better tools.

With a study on a list of 71 tools, we have drawn a map of the main characteristics of existing tools, their pros and cons, the assumptions to be matched to use them, the information they gather, the techniques they use to efficiently collect data, the way in which they present the results, and several others features. The result is meant to be useful for both researchers and practitioners (e.g., DevOps engineers, as well as tool vendors) working in this area. This audience can get feedback to tackle new research challenges, e.g., pertaining to overhead reduction and quality improvement, to use the map and given suggestions to select or customize the tools for the system being developed/operated, to integrate the tool with other facilities (such as for testing and issue management). The full list of classified tools is on the replication package[11] and on Zenodo (uDEVOPS2020, 2023).

CRediT authorship contribution statement

L. Giamattei: Conceptualization, Methodology, Investigation, Software, Formal analysis, Validation, Writing, Visualization, Data curation. **A. Guerriero:** Conceptualization, Methodology, Investigation, Formal analysis, Validation, Writing, Visualization, Data curation. **R. Pietrantuono:** Conceptualization, Methodology, Supervision, Funding acquisition, Project administration, Writing, Visualization. **S. Russo:** Methodology, Supervision, Writing, Visualization. **I. Malavolta:** Conceptualization, Methodology, Investigation, Supervision, Software, Formal analysis, Validation, Writing, Visualization, Data curation. **T. Islam:** Formal analysis, Data curation, Validation, Writing, Visualization. **M. Dînga:** Formal analysis, Data curation, Visualization. **A. Koziolek:** Methodology, Supervision, Validation. **S. Singh:** Methodology, Investigation, Formal analysis, Validation, Writing, Data curation. **M. Armbruster:** Methodology, Investigation, Formal analysis, Validation, Writing, Data curation. **J.M. Gutierrez-Martinez:** Methodology, Investigation, Formal analysis, Validation, Writing, Data curation. **S. Caro-Alvaro:** Methodology, Investigation, Formal analysis, Validation, Writing, Data curation. **D. Rodriguez:** Methodology, Investigation, Formal analysis, Validation, Writing, Data curation. **S. Weber:** Methodology, Investigation, Formal analysis, Validation, Writing, Data curation. **J. Henss:** Methodology, Supervision, Validation. **E. Fernandez Vogelin:** Formal analysis, Data curation. **F. Simon Panojo:** Formal analysis, Data curation.

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.

Data availability

I have shared the link to data/code, on GitHub (https://github.com/uDEVOPS2020/Monitoring-Tools-for-DevOps-and-Microservices-a-Systematic-Study) and on zenodo: https://doi.org/10.5281/zenodo.8212052.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 871342 "uDEVOPS".

References

Bento, A., Correia, J., Filipe, R., Araujo, F., Cardoso, J., 2021. Automated analysis of distributed tracing: Challenges and research directions. J. Grid Comput. 19, 1–15.

Di Francesco, P., Lago, P., Malavolta, I., 2019. Architecting with microservices: A systematic mapping study. J. Syst. Softw. 150, 77–97.

Di Francesco, P., Malavolta, I., Lago, P., 2017. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: 2017 IEEE International Conference on Software Architecture. ICSA, pp. 21–30.

Ebert, C., Gallardo, G., Hernantes, J., Serrano, N., 2016. DevOps. IEEE Softw. 33 (3), 94–100.

Firtman, M., 2018. Hacking Web Performance. O'Reilly Media, Inc., Sebastopol, CA, USA.

Fleiss, J.L., Levin, B., Paik, M.C., 2003. The measurement of interrater agreement. In: Statistical Methods for Rates and Proportions. John Wiley & Sons, Ltd, pp. 598–626. http://dx.doi.org/10.1002/0471445428.ch18.

Garousi, V., Felderer, M., Mäntylä, M.V., 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Inf. Softw. Technol. 106, 101–121.

Ghofrani, J., Lübke, D., 2018. Challenges of microservices architecture: A survey on the state of the practice. In: CEUR Workshop Proceedings. Vol. 2072, pp. 1–8.

Haselböck, S., Weinreich, R., Buchgeher, G., 2018. An expert interview study on areas of microservice design. In: 2018 IEEE 11th Conference on Service-Oriented Computing and Applications. SOCA, pp. 137–144. http://dx.doi.org/10.1109/SOCA.2018.00028.

Hernantes, J., Gallardo, G., Serrano, N., 2015. IT infrastructure-monitoring tools. IEEE Softw. 32 (4), 88–93.

Huye, D., Shkuro, Y., Sambasivan, R.R., 2023. Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows. In: 2023 USENIX Annual Technical Conference. USENIX ATC 23, USENIX Association, pp. 419–432.

Jabbari, R., bin Ali, N., Petersen, K., Tanveer, B., 2016. What is DevOps? A systematic mapping study on definitions and practices. In: Proceedings of the Scientific Workshop Proceedings of XP2016. In: XP '16 Workshops, Association for Computing Machinery, New York, NY, USA.

Kitchenham, B., Brereton, P., 2013. A systematic review of systematic review process research in software engineering. Inf. Softw. Technol. 55 (12), 2049–2075.

Knoche, H., Hasselbring, W., 2019. Drivers and barriers for microservice adoption – A survey among professionals in Germany. Enterpr. Model. Inf. Syst. Archit. - Int. J. Concept. Model. 1–35.

Lewis, J., Fowler, M., 2014. Microservices - a definition of this new architectural term. Available at: http://martinfowler.com/articles/microservices.html.

Petersen, K., Vakkalanka, S., Kuzniarz, L., 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. Inf. Softw. Technol. 64, 1–18.

Richardson, C., 2018. Microservices Patterns: With Examples in Java. Manning.

Riet, J.V., Malavolta, I., Ghaleb, T.A., 2023. Optimise along the way: An industrial case study on web performance. J. Syst. Softw. 198, 111593.

Rothstein, H.R., Hopewell, S., 2009. Grey literature. In: The Handbook of Research Synthesis and Meta-Analysis. Vol. 2, pp. 103–125.

Schroeder, B.A., 1995. On-line monitoring: A tutorial. Computer 28 (06), 72–78.

Shkuro, Y., 2019. Mastering Distributed Tracing: Analyzing Performance in Microservices and Complex Systems. Packt Publishing Ltd.

Soldani, J., Tamburri, D.A., Van Den Heuvel, W.-J., 2018. The pains and gains of microservices: A systematic grey literature review. J. Syst. Softw. 146, 215–232.

TIOBE, 2022. TIOBE Index. [Online]. (Accessed 17 July 2023). URL: https://www.tiobe.com/tiobe-index.

uDEVOPS2020, 2023. uDEVOPS2020/Monitoring-Tools-for-DevOps-and- Microservices-a-Systematic-Study: Release 1. http://dx.doi.org/10.5281/zenodo.8212052.

Verdecchia, R., Lago, P., Ebert, C., De Vries, C., 2021. Green IT and green software. IEEE Softw. 38 (6), 7–15.

Viggiato, M., Terra, R., Rocha, H., Valente, M.T., Figueiredo, E., 2018. Microservices in practice: A survey study. ArXiv abs/1808.04836.

Wang, Y., Kadiyala, H., Rubin, J., 2021. Promises and challenges of microservices: an exploratory study. Empir. Softw. Eng. 26 (4), 63.

Waseem, M., Liang, P., Shahin, M., 2020. A systematic mapping study on microservices architecture in DevOps. J. Syst. Softw. 170, 110798.

Waseem, M., Liang, P., Shahin, M., Di Salle, A., Márquez, G., 2021. Design, monitoring, and testing of microservices systems: The practitioners perspective. J. Syst. Softw. 182, 111061.

Zhang, H., Li, S., Jia, Z., Zhong, C., Zhang, C., 2019. Microservice architecture in reality: An industrial inquiry. In: 2019 IEEE International Conference on Software Architecture. ICSA, IEEE, pp. 51–60.

Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., Ding, D., 2021. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. IEEE Trans. Softw. Eng. 47 (2), 243–260.

Luca Giamattei (Ph.D. student) received the M.S. degree in computer engineering in 2021 from the Federico II University of Naples, Italy. He is currently a Ph.D. student in Information Technology and Electrical Engineering from the same university. His main research interests are in testing of DNN-enabled systems and distributed software systems. In this context, he collaborated in international projects. He published in international conferences and journals in the field of software engineering and software testing.

Antonio Guerriero (Ph.D.) is Assistant Professor at Federico II University of Naples, Italy. He received the Ph.D. degree in Information Technology and Electrical Engineering from the same university in 2022. His main research interests are in testing of ML-based systems and distributed software systems. In this context, he collaborated in both national and international projects. He published in international conferences and journals in the field of software reliability, software engineering, and software testing.

Roberto Pietrantuono (Member of ACM, Senior Member of IEEE) is an Associate Professor with the University of Naples Federico II. Since 2007, he has been with the Dependable Systems and Software Engineering Research Team. His research interests are in the area of software engineering, particularly software testing and dependability of software (and AI-based) systems. He regularly serves on program committees of international conferences and journals and Associate Editor of IEEE Transactions on Services Computing and of Software Quality Journal. He is involved in several projects and currently coordinates an MSCA RISE European Project (μDevOps).

Stefano Russo (Ph.D.) is Professor of Computer Engineering at Federico II University of Naples, Italy, where he teaches Software Engineering and Distributed Systems, and leads the DESSERT research group (www.dessert.unina.it). He (co-)authored over 200 papers in the areas of software testing, software aging, middleware technologies, mobile computing. He is Associate Editor of IEEE Transactions on Services Computing, and Senior Member of the IEEE.

Ivano Malavolta is an Associate Professor in the Software and Sustainability research group and Director of the Network Institute at the Vrije Universiteit Amsterdam, The Netherlands His research interests include empirical software engineering, with a special emphasis on software architecture, robotics software, and energy-efficient software. He authored more than 150 scientific articles in peer-reviewed international journals and international conference proceedings. He is program committee member and reviewer of international conferences and journals and Associate Editor of IEEE Software, the International Journal of Robotics Research, and the Frontiers in Robotics and AI journal. He received a Ph.D. in computer science from the University of L'Aquila, Italy. He is a Member of IEEE, ACM, VERSEN, and Amsterdam Data Science. More information about Ivano is available on his official website: https://www.ivanomalavolta.com.

Tanjina Islam received her BSc. in Computer Science and Engineering from BRAC University, Dhaka, Bangladesh, in 2014. In 2019, she obtained her MSc. in Computer Science, specializing in Software Engineering and Green IT, through a joint degree program offered by Vrije University Amsterdam (VU) and the University of Amsterdam (UvA). Following her studies, Tanjina worked as a Junior Docent with the Software and Sustainability (S2) research group at Vrije University Amsterdam. Before embarking on her academic pursuits, she gained three years of valuable experience as a Software Engineer across various IT industries in Bangladesh. Currently, Tanjina is a Ph.D. candidate within the Complex Cyber Infrastructure (CCI) research group at the University of Amsterdam's Informatics Institute. Her research focuses on investigating the energy consumption and security aspects of machine learning applications in the cloud-to-edge continuum.

Madalina Dînga is a software engineer interested in cloud native development and software sustainability. She received a MSc in Computer Science from Vrije Universiteit Amsterdam in 2022, specialising in Software Engineering and Green IT. Her MSc Thesis delved into an empirical assessment of the energy and performance overhead of monitoring tools on microservices.

Anne Koziolek is a full professor at the Karlsruhe Institute of Technology (KIT), Germany. Her research interests are software architecture, model-based quality prediction, architecture recovery, and recently the use of LLMs in supporting all tasks related to software architecture. In all these efforts, she is interested in reconciling agile, code-centric software development with model-based software engineering. Anne received her Diplom degree in informatics from the University of Oldenburg in 2007 and her Ph.D. from KIT in 2011. Following her doctorate, she was a postdoc researcher at the University of Zurich until 2013. She is a member of the ACM, IEEE, and GI. More information about Anne is available at https://mcse.kastel.kit.edu/staff_Koziolek_Anne.php.

Snigdha Singh, is a doctoral researcher at Karlsruhe Institute of Technology, Germany, in the chair of Modelling for Continuous Software Engineering. She is a Masters in Computer Science from IIIT Delhi, India. Her research interests are reverse engineering for architecture extraction with a focus in asynchronous communication between Microservices and architecture recovery of message based systems. More information about her is available at https://mcse.kastel.kit.edu/staff_Snigdha_Singh.php

Martin Armbruster is a doctoral researcher at the Modelling for Continuous Software Engineering (MCSE) group at the Karlsruhe Institute of Technology (KIT), Germany. His research and research interests focus on model-driven software development, in particular, code modeling, the model-driven and continuous extraction of architectural performance models from code, and consistency management of models. Martin received his master's degree in Informatics from KIT in 2021. More information can be found at https://mcse.kastel.kit.edu/staff_martin_armbruster.php.

José-María Gutierrez-Martinez, Ph.D. in Computer Science, is an associate professor at the University of Alcalá, head of the Computer Science Department for 5 years and coordinator of a Master Program in Agile Web Development. He has extensive works on mobile devices and their integration with traditional information systems and has been working also with for e-learning and m-learning technological enhancements. He has participate in research projects for e-health and lead some of them about the use of mobile devices to help with psychiatric treatments, medicine doses, wheelchair use and tablet adaptations to highly dependent users. Some of these projects include de use of sensors and microcontrollers and their integration with mobile phones applications. He has worked in 7 European funded projects and many local public funded research projects. He also encourages the relation with companies, been the head of a research chair that uses mobility and IA to help industrial installations management.

Sergio Caro-Alvaro is an Assistant Professor in the Computer Science Department at University of Alcala (Spain). He received his Ph.D. degree in the Computer Science Department of the University of Alcala (Spain). His research interests include usability, user experience and gamification.

Daniel Rodriguez is currently an associate professor at the Computer Science Department of the University of Alcala, Madrid, Spain. He is also a regular visiting researcher at Oxford Brookes University, UK. Also, he was a lecturer at the University of Reading, UK. Daniel earned his degree in Computer Science at the University of the Basque Country (EHU) and Ph.D. degree at the University of Reading, UK. His research interest include the application of data mining and optimization techniques to Software Engineering problems in particular.

Sebastian Weber studied computer science at the Karlsruhe Institute of Technology (KIT) and received his degree in 2022. He joined the software engineering and quality department at the FZI Research Center for Information Technology afterwards and is a doctoral student of Professor Reussner at the Software Design and Quality (SDQ) group at KIT. His research interests focus on the analysis and modeling of complex and heterogeneous systems through analysis composition and multi-level simulation.

Jörg Henss studied computer science at the University of Karlsruhe. After completing his degree in 2008, he joined the Software Design and Quality (SDQ) group at Karlsruhe Institute of Technology in 2009. Since 2015 he is head of the software engineering and quality department at the FZI Research Center for Information Technology. His research interests include interoperability of simulations, model-driven software development (MDSD), and modern software development in the context of operational and mobile application systems.

Estrella Fernández Vogelin is a computer scientist and electronic systems engineer, with master degree in cyber security and privacy. Her background goes from hardware design and testing to software quality, having worked for avionics hardware and working in the present for Panel Sistemas Informáticos as a software engineer in test for client Iberia and collaborating with Vrije University in microservices research.

Fernando Simön Panojo, graduate in Physical Sciences from the Complutense University of Madrid, specializing in physical devices and control. He worked in several consulting on development and quality projects until finally starting to work in 2007 at "Panel Sistemas Informáticos", a company where he is currently working for the client Iberia, developing QA (Quality Assurance) tasks in both manual and automated tests. and system monitoring. In 20222 he began to collaborate (from "Panel Sistemas Informáticos") with the Vrije University of Amsterdam in the study of systems and application monitoring tools.