Project funded by the EU Horizon 2020 programme under the Marie Skłdowska-Curie grant agreement No 871342

uDevoOps

Software Quality Assurance for Microservice Development Operations Engineering

Deliverable D2.1. Analysis and characterization of Microservice Development Operations Engineering and Modelling techniques and formalisms for µDevOps

November 2023

Abstract

This deliverable reports about the activities carried out in work package 2. The goal is to have a characterization of the context in which microservices developed according to a DevOps paradigm operate. The characterization of the context is expected to support quality-assurance activities (particularly, testing activities) in what we called context-driven testing.

Since a prominent feature of μDevOps development of microservices is the massive use of field data as feedback. Such data , gathered though continuous monitoring, are the best way to characterize the context, as they can tell about what is the running architecture (e.g., which services and which version are deployed), how they are interacting, what is their failing behaviour (how often they fail, and how), and how the user is exercising the system . Our plan is to exploit all these data to parameterize models to support several quality-related tasks, such as testing, root cause analysis, fault prediction, performance and energy analysis and optimization, etc.

To this aim, we need two main features: *monitoring* and *modelling*. This deliverable explores both these aspects. We first conduct an extensive study about monitoring tools in the microservice/DevOps context; then we define possible modelling strategies to capture the concepts relevant for the subsequent phases.

Contents

1 Preliminaries

1.1 Microservices Architecture

Microservice architecture (MSA) is a software architectural style which is gaining popularity in many companies [33]. Netflix, eBay, Amazon, Twitter, PayPal and many other web-based services have evolved to this paradigm recently. MSA shifts traditional service-oriented architectures (SOA) from a share-as-much-as-you-can philosophy, focused on reuse, to a share-nothing philosophy, emphasizing strong service decoupling.

MSA applications are built by architecting a set of services, each providing a well-defined and self-contained business capability and high independence from others. They are usually developed according to the API Gateway pattern, where "like a facade, an API gateway encapsulates the application's internal architecture and provides an API to its clients; it may also have other responsibilities, such as authentication, monitoring, and rate limiting" [72].

Combined with technologies such as RESTful protocols and containers and agile development practices such as DevOps, MSA features lightweight communication and independent and rapid service deployment. These characteristics promote scalability, flexibility, maintainability, prompt reaction to changes and failures, and frequent software releases.

1.2 DevOps

The term DevOps (development and operations) indicates "a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality" [7].

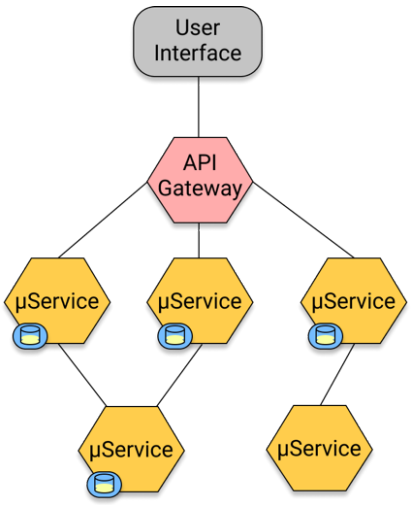

Figure 1: Microservice Architecture

DevOps cycle is composed by many phases, each one with a different purpose [45]. According to Virmani (IBM) [90], DevOps is made by the following phases:

- Continuous Planning: it allows a continuous feedback channel with customers in order to adjust the business plan if needed;
- Continuous Integration: it tries to integrate early the changes in the system sharing them with the team, and allowing to continuously validate the behavior;
- Continuous Deployment: it aims to reduce the delay in the software delivery automatizing the deployment and provisioning of hardware and various cloud providers;
- Continuous Testing: complementary to Continuous deploy, it aims to automate each testing process in order to make the software delivery faster;
- Continuous Monitoring: it observe various quality parameters throughout and hence ability to react to any surprises in timely manner.

The overall goal is "context modelling" to enable context-driven testing of MSA - a way we aim to follow is AIOps in MSA, to enable MSA improvement in terms of testing, deployment, quality assurance, and so on. The deliverable should explore the state-of-the-art on topics related to context modelling (in particular in MSA): this concerns *monitoring* strategies and tools, and then *modelling*, with these main questions: **WHAT IS MONITORED, HOW MONITORING IS DONE, IF/HOW MODELLING IS USED**. As for modelling dimensions we considered: usage (user interaction), failures, architecture (and how it changes), behaviour (e.g., interaction between MS).

2 Monitoring in DevOps and MSA

It is well-known in both academia [22] and practice [94] that developing and operating microservice-based systems is a difficult task, mainly due to their distributed nature, complex and dynamic deployments, and technological heterogeneity [78]. 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 [93].

Various *monitoring tools* are currently widely-used by DevOps teams for collecting, aggregating, and analysing metrics in order to give meaningful insights about the system's overall health and behaviour 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 analyse them, primarily with the goal of alerting DevOps teams when a relevant signal is detected, so that they can take corrective actions [26, 43]. 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, etc. As an indication, a recent study targeting microservices practitioners identified 23 different monitoring tools used by practitioners when monitoring microservices systems [94]. 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.

2.1 Objective

The **objective** of our 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.

2.2 Definitions

Monitoring. *Monitoring* is the process of dynamically gathering, interpreting, and elaborating data about the execution of the system under test (SUT) [75].

We distinguish *direct* and *indirect* monitoring, based on the source of information, that is, who produces the information. Monitoring is *direct* if the monitored information comes directly from the SUT, *indirect* if the monitored information comes from the software, physical or human environment where the SUT operates, for instance OS resources consumption, data read by sensors or data about user interactions with the system.

The activities for gathering information include:

- *logging*: the process of recording textual and/or numerical information about events of interest
- *tracing*: the process of recording information about the control flow of a SUT during its execution.

Logging and tracing differ in their goal, even when implemented with similar techniques, for instance by instrumentation: Tracing records the execution flow of the SUT execution without referring to specific classes of events of interest, while logging focuses on the events of interest, for instance, recording errors or failures

Monitoring metrics. A summary of the key metrics for microservices architectures is provided by Fowler [32] and Waseem *et al.* [94]. Key metrics can

[1] `https://prometheus.io`
[2] `https://www.jaegertracing.io`
[3] `https://www.elastic.co/elasticsearch`

be divided in two sets: host and infrastructure metrics, and microservice metrics. Host and infrastructure metrics are those that pertain to the status of the infrastructure and the servers on which the microservice is running, while microservice metrics are metrics that are unique to the individual microservice. Host and infrastructure metrics are:

- CPU utilized by the microservice on each host;
- RAM utilized by the microservice on each host;
- available threads;
- microservice's open file descriptors (FD);
- number of database connections that the microservice has to any databases it uses.

Key metrics at the microservice level can depend on the language used to implement the microservice, in general the following metrics can be considered:

- language-specific key metrics;
- availability of the microservice;
- service-level agreement (SLA) of the service;
- latency (of both the service as a whole and its API endpoints);
- success of API endpoints;
- responses and average response times of API endpoints;
- services (clients) from which API requests originate (along with which endpoints they send requests to);
- errors and exceptions (both handled and unhandled);
- health and status of dependencies.

Monitoring practices. Waseem *et al.* [94] identified six different monitoring practices from the grey literature [16, 70, 80]:

- log management: the process for managing event logs (generation, transmission, storage, analysis, disposition);
- exception tracking: handled and unhandled exceptions are gathered in a log with their exception traces;
- health check API: used to get the operational status, performance, and dependencies of microservices;
- log deployment: logging of the microservices during deployment phase;

- audit logging: recording event logs, regarding a sequence of activities or a specific activity;
- distributed tracking: monitoring the microservices behavior monitoring the tracing all the involved services.

These monitoring practices are enabled by a set of observability patterns, as depicted in Figure 2 [73].

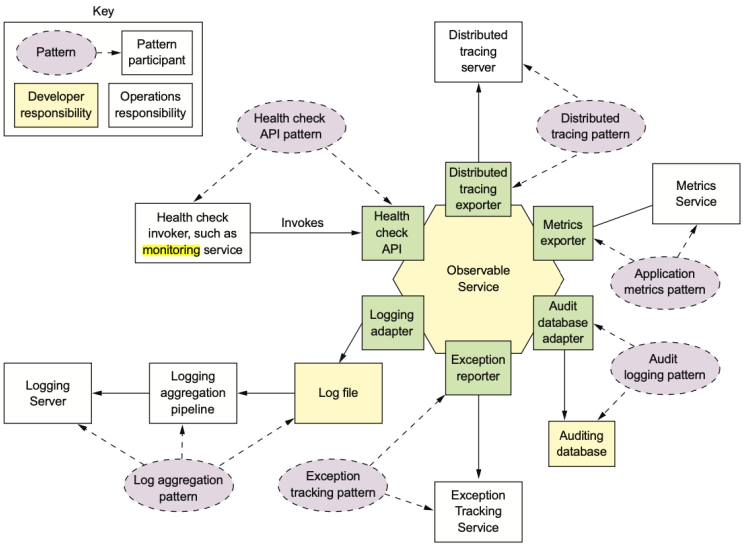

Figure 2: Obsevability patterns [73]

Monitoring tools. Waseem *et al.* [94] classify tools for monitoring into two categories: *libraries* and *platforms*. Libraries are used during the development of microservices and permit collecting the application data. Platforms allow gathering and analyzing data from different sources, such as the hosts, infrastructure, and microservices [80].

Monitoring challenges. According to Waseem *et al.* [94] we report the following challenges:

- MC1 Collection of monitoring metrics data and logs from containers
- MC2 Distributed tracing
- MC3 Having 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 inside containers
- MC9 Maintaining monitoring infrastructures.

2.3 Related research

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

2.3.1 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 [94], 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 [53], 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 [89]. 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 [35]. These are clearly related to the challenges of monitoring we considered in this report (identified in [94]).

Another survey with 21 participants was conducted by Wang *et al.* [92], 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 [42, 99, 101].

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.

2.3.2 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 [33], 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 [22].

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 [78]. 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 report.

Another work by Waseem [93], 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.

2.4 Search process

2.4.1 Research questions

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.* [94].
- **RQ2.** *What information is gathered to characterize the behaviour 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.

2.4.2 Tools selection process

To address the above questions, the research is conducted according to the protocol shown in Figure 3.

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.

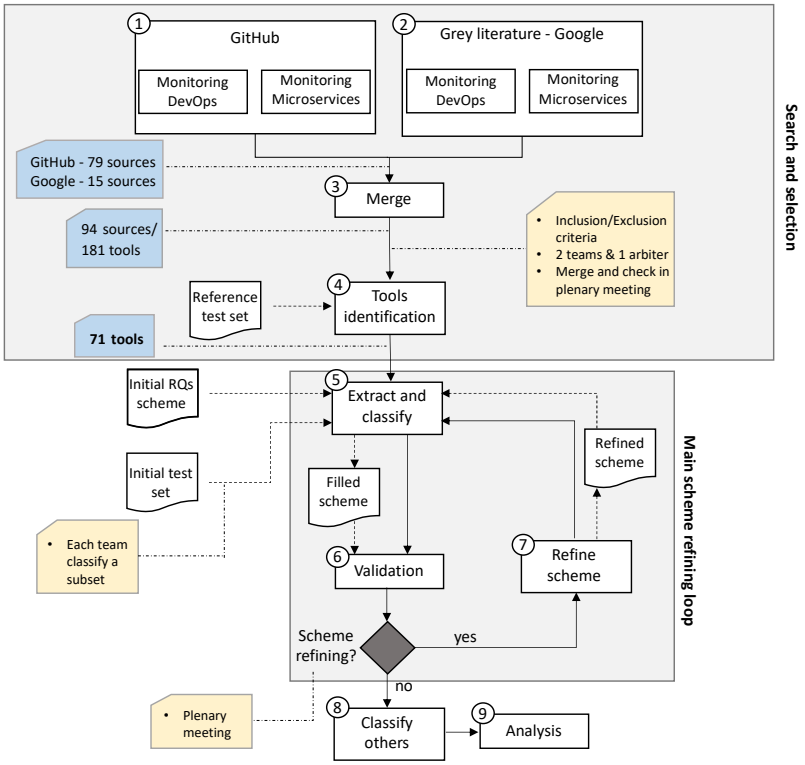

Figure 3: Protocol

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

By merging the results of these searches ③, 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;

  - 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)[4]. 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.

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

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 the common interpretation in literature [31], 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 [66], [52]. 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.

2.4.3 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 Figure 3), 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., [94]. To refine and agree on the scheme, we used the *initial test set* approach [66]: 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 favoured the detection of inconsistent classifications

Table 1: Analysed monitoring tools

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

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.

2.4.4 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 (Section 2.5-2.8). Then, a cross-cutting analysis relating sub-dimensions was also performed in order to highlight interesting patterns in the characteristics of the tools (Section 2.9).

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

2.5 Results - Overview

Figure 4 shows the number of selected (included/excluded) tools by source type.[5] 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 analyse 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

[5]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]

2021 or 2022.

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`.

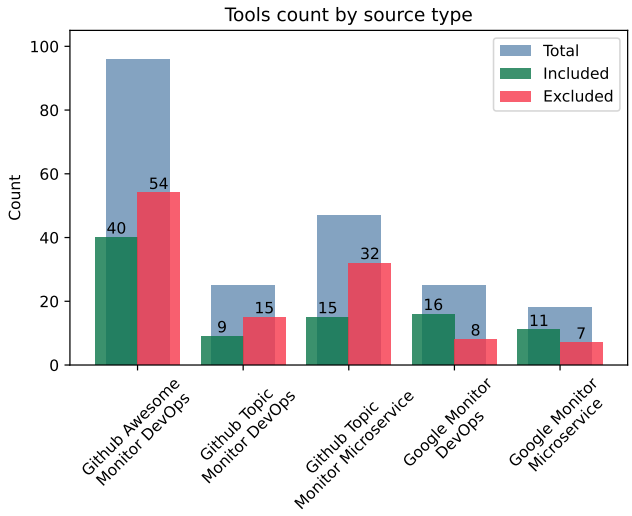

Figure 4: Tools selection count

2.6 Results – Functional and Technological Features. Addressed Challenges

2.6.1 Targets, Features, Motivation

We have analysed 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 Figure 5, 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 Figure 6, the tools clearly all have collection and monitoring services (in form of traces, logs and 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

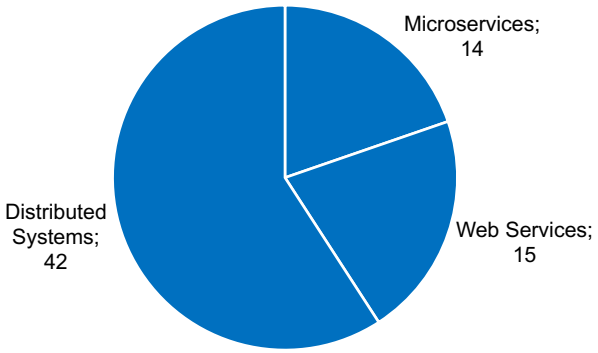

Figure 5: Tools target

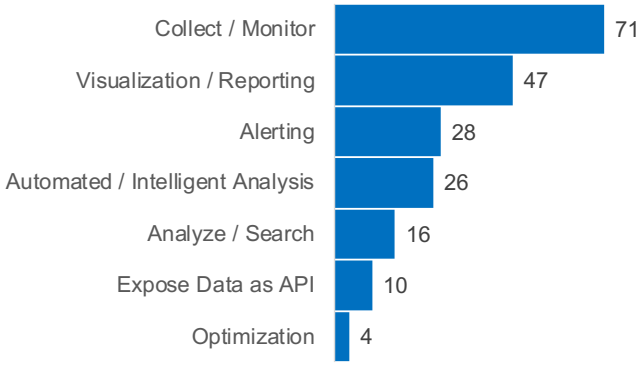

Figure 6: Tools' Features/Motivation

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.

2.6.2 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.

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[6] 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.

2.6.3 Technologies

2.6.4 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 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.

Required technologies. Figure 7 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

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

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

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

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.

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 instrumenta-

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

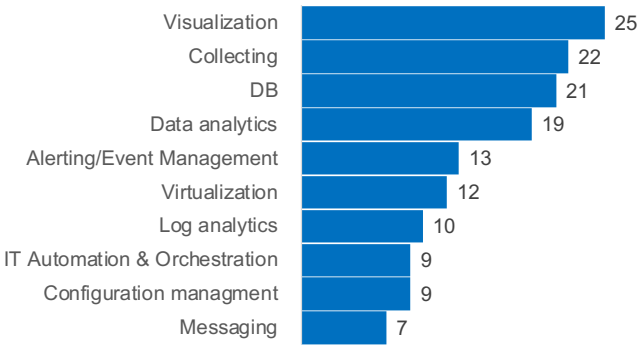

Figure 7: Tools required technologies

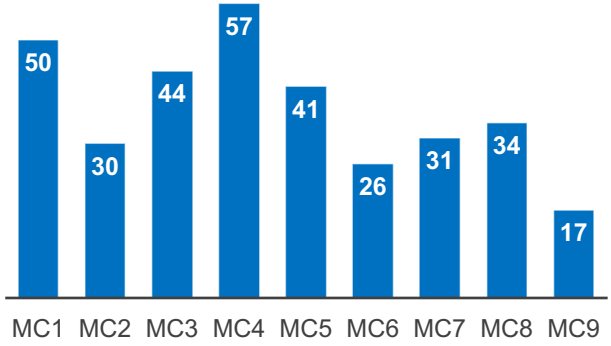

Figure 8: Challenges addressed

tion. 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.

2.6.5 Addressed challenges

We analyze if and how the tools address the list of challenges as identified by Waseem et *al.* [94]. The challenges are summarized in Section 2.2.

The most important challenges addressed by the tools, as shown in Figure 8, 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.* [94], and are related to (i) the communication between

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

hundreds of microservices (hence referring to complexity of MSA), (ii) absence of a standardized infrastructure for run-time monitoring (hindering collection of data), (iii) different languages, databases, and frameworks for developing microservices, and (iv) logs and dataflows in different format.

Indeed, we found that several of the analysed 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 2.9.

2.7 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.

2.7.1 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.

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*

Table 8: 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

metrics. Besides, Elastic Search also covers *Health* and *User-sessions* metrics.

2.7.2 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 9, 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.).

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.

2.7.3 Distributed tracing

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

As shown in Table 10, 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.

2.7.4 Failures/events logging

This parameter investigates whether the tools support failures/events logging or not. As shown in Table 11, 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.

Table 9: 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

Table 10: 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 11: 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

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.

2.7.5 Targeted quality attribute

This parameter is about the quality attributes explicitly targeted by the monitoring tools. As shown in Table 12, 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 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 12: 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

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 13. 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.

Table 13: 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

2.8 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.

Table 14: 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

2.8.1 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 15 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: 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

2.8.2 Monitoring patterns and practices

Table 14 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 14. 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.

2.8.3 Monitoring Granularity

Table 16: 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

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 infrastruc-

Table 17: 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

ture. The mapping of tools to the levels is displayed in Table 16. 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 17 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.

2.8.4 Integration with Testing

Table 18: 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

Regarding testing, we looked if the monitoring tools support testing in some way. As shown in Table 18, 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 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.

2.9 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 2.9.1), (ii) open challenges to be addressed by both researchers and tool vendors (Section 2.9.2), and (iii) cross-cutting findings emerging from our orthogonal analysis (Section 2.9.3).

Table 19: 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.	2.5
Most 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%).	2.6.1
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.	2.6.2
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.	2.6.3
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** [94] addressed by more than 50% of the tools are those related to performance (MC4), metrics and logs (MC1), complexity (MC3), and data analysis (MC5).	2.6.5
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).	2.7
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).	2.8
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.	

2.9.1 Main findings and guidance for DevOps engineers

Table 19 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.

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)?*
  - 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.

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.

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 pos-

sible 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.

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 [10, 44]. However, distributed tracing tends to be more complex and resource-demanding than the collection of individual metrics for each service [76]; this means that potentially distributed tracing might lead to a higher overhead for the system being monitored [10]. 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.

Instrumentation. As shown in Table 19, 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[7]) and (ii) allocate proper time and resources towards the **co-evolution of the microservices source code and their instrumentation code**.

Community support. Finally, also the state of the community around the chosen monitoring tool plays a strong role. Some of the analysed 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 com-

[7] `https://opentelemetry.io/ecosystem/integrations`

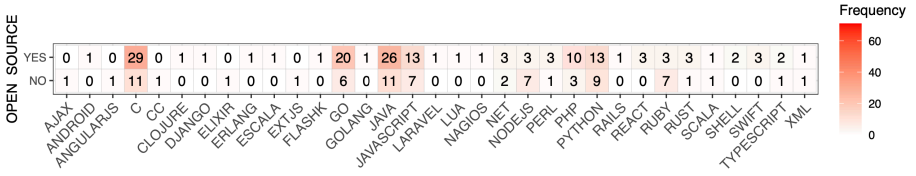

Figure 9: Co-occurrences between *open§/closed source* and *programming language*

panies 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.

2.9.2 Open challenges for researchers and tool vendors

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 analysing 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, 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.

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 [94].

- **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 [88] 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.

2.9.3 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 2.4.3). 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[8].

Open source and programming languages. Figure 9 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 [84]. 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.

Open source and addressed challenges. We crosschecked the challenges addressed by each monitoring tool and whether it is an open-source project or

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

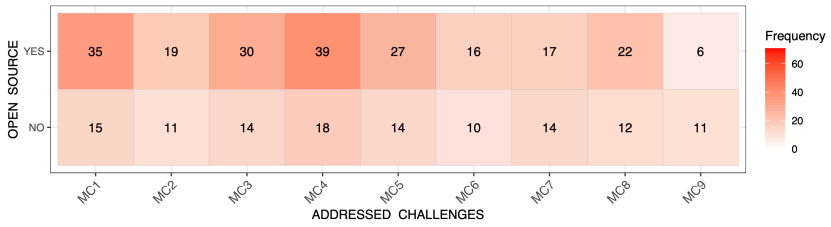

Figure 10: Co-occurrences between *open§/closed source* and *addressed challenges*

not. Figure 10 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, 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.

User-oriented metrics and system-oriented metrics. Figure 11 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 analysing the co-occurrences of user-oriented metrics (see Section2.9.3) and system-oriented metrics (see Section2.9.3).

Co-occurrences of user-oriented metrics. Figure 12 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.,

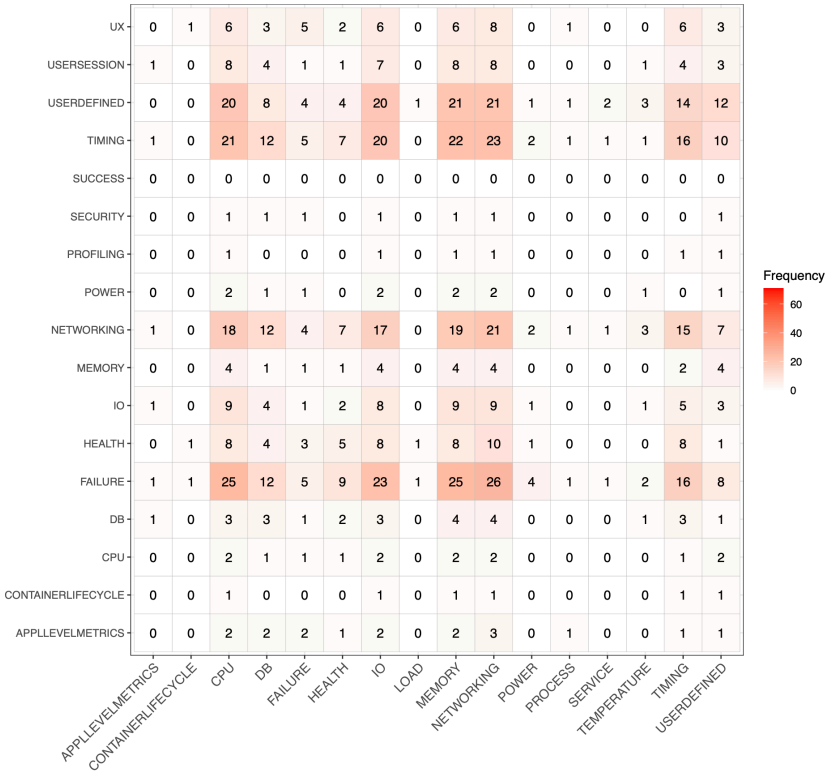

Figure 11: Co-occurrences between *user-oriented* and *system-oriented metrics*

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" [30]. 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 [74]. 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.

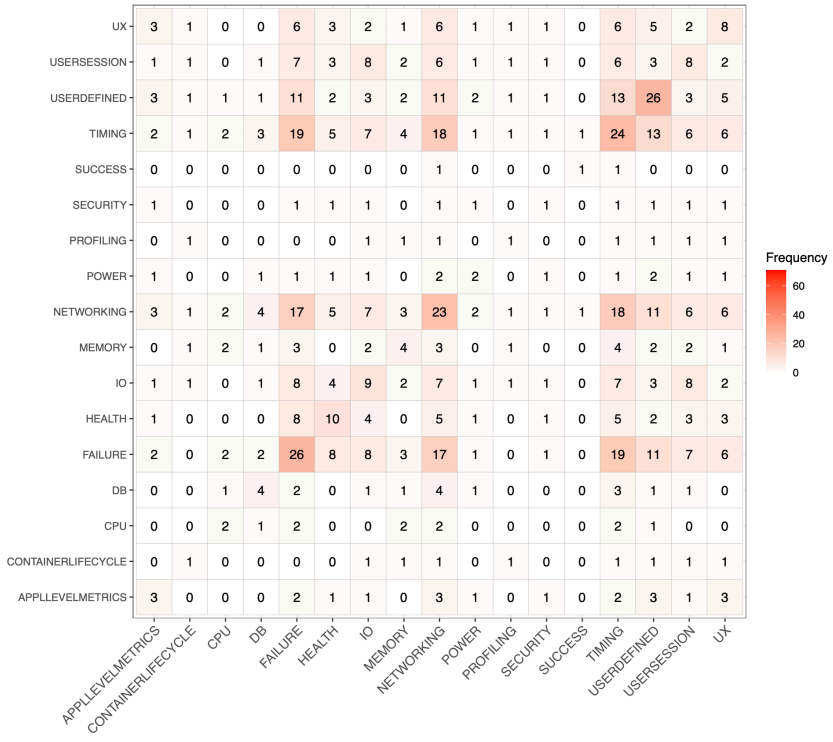

Figure 12: Co-occurrences between different *user-oriented* metrics

Co-occurrences of system-oriented metrics. Figure 13 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 [88], 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 consump-

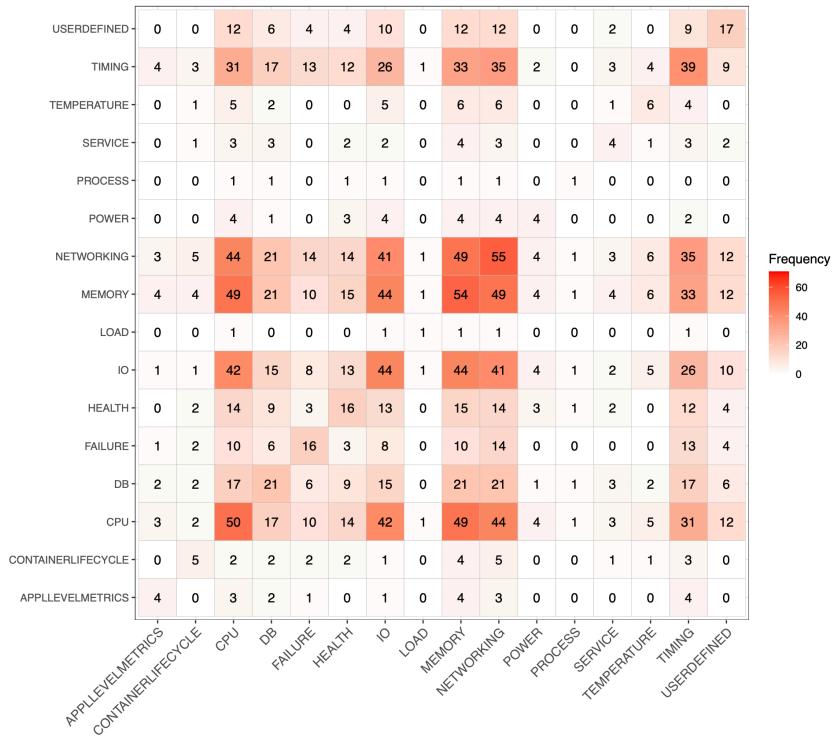

Figure 13: Co-occurrences between different *system-oriented* metrics

tion 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`[9]. `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.

Requests tracing and targeted quality attributes. Figure 14 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**

[9] `https://github.com/kube-green/kube-green`

Figure 14: Co-occurrences between the *support for tracing* and *quality attributes*

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 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.*

Testing and targeted quality attributes. Figure 15 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

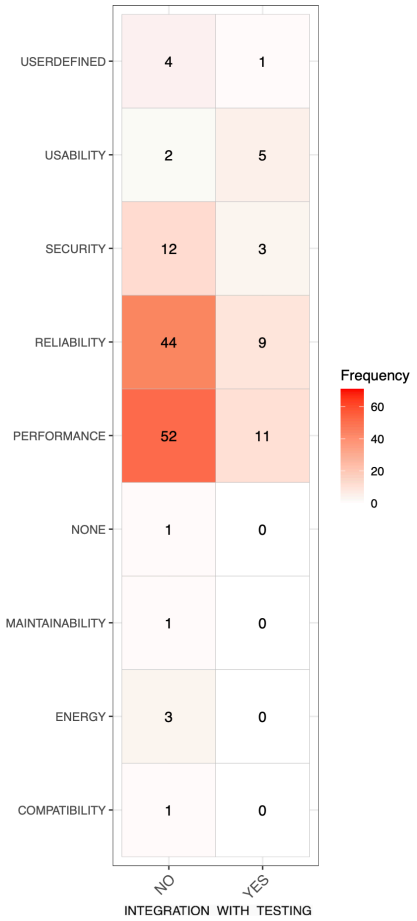

Figure 15: Co-occurrences between the *integration with testing* and *quality attributes*

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 [88], 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).

3 Modeling

A system designer has in his or her possession a wide range of analytical modelling techniques to choose from. Each of these techniques has its own strengths and weaknesses in terms of accessibility, ease of construction, efficiency and accuracy of solution algorithms, and availability of supporting software tools. The most appropriate type of model depends upon the complexity of the system, the questions to be studied, the accuracy required, and the resources available to the study. In the following, many references about usage modeling and failure modeling are analyzed to understand their applicability for MSAs in a DevOps context.

3.1 Usage modeling

Modeling how a system is used in operation is a fundamental practice to understand the quality provided by that system after the deplyment in the execution environment. This Section reviews strategies for usage modelling proposed and used in the context of Microservice a application and/or in a DevOps context.

When evaluating distributed software system quality the concept of operational profile assumes a critical importance as a quantitative characterization of how customers will use the system in production [62]. In fact, to satisfy user requirements it is important to be able to profile actual usage, so that the system can then be tested by reproducing and predicting users' experience. It is well known that deriving an operational profile before product release is hard, and the overhead costs often discourage SRE adoption in industry [56].

The continuous monitoring of the system during the operation proper of DevOps practices allows a more cheap definition of an operational profile [11,68].

In the context Microservices Arichitectures (MSA) it is defined as probability distribution over the demand spaces of: all microservices [67], or different system's users [13]. A more fine modeling can be performed considering stochastic models, like Markov models in order to represents users behavior or the system itself [85, 86, 95].

Pietrantuono *et al.* [67] model the usage of Microservices through a probability distribution over the demand spaces of all microservices. In this way the usage of a MSA can be described at methods granularity. That representation of the usage is called *operational profile*. The operational profile can be used for many purpose, like reliability testing [67, 69] or performance/load testing [1]. In this latter case, the operational profile can also be seen with a different concern, related to the different characteristics of the operational environment in terms of phisical/virtual resources (CPU, Memory, and so on).

Camilli *et al.* [13] gives a probabilistic representation of user sessions in terms of a Discrete Time Markov Chain (DTMC). In particular, they extend the modeling approach introduced in [91] by additionally considering the input space in the construction of the Markov chain. Essentially, the nodes of the DTMC

model represent the requests that can be issued to the system by providing either a *valid* or *invalid* input values, according to the API specification. Thus, the input space for each request is partitioned into valid and invalid classes, henceforth referred to as *request classes*. The transitions (i.e., weighted edges) in the DTMC specify the probability of moving from a given request class to the next one.

3.2 Failure modeling

This Section reviews strategies that are suitable for capturing and modelling the failing behaviour of a Microservice application in a DevOps context. Also, we report about those works that have applied such strategies to microservices in the last years.

Combinatorial models. Combinatorial models are the simplest ones, as do not make use of any state-space representation. In these models, the measures of interest about the quality of the overall system are derived by combining the corresponding measures at component-level assuming the independence between components. The system is typically divided into a set of non-overlapping modules, each one associated with either a probability of working, P_i (or a probability as function of time, e.g., $R_i(\mathrm{t})$). The goal is to derive the overall P_{sys} value (or function $R_{sys}(\mathrm{t})$), representing the probability that the system survives (until t). These models typically enumerate all the system states, by using combinatorial counting techniques to simplify the process.

Typical metrics of interest are reliability, availability, survivability or resiliency. Reliability block diagrams (RBDs), fault trees (FTs) and reliability graphs (RGs) are combinatorial formalisms commonly used to study the dependability of systems. They are concise, easy to understand, and have efficient solution methods. However, realistic features such as interrelated behaviour of components, imperfect coverage, non¬zero reconfiguration delays, and combination with performance cannot be captured by these models. These arguments led to the development of new formalisms, such as dynamic fault trees (DFTs) and dynamic reliability block diagrams (DRBDs), to model reliability interactions among components or subsystems. A brief overview of traditional non-state space models can be found in [63], while some of their "dynamic" extensions are outlined in [23], such as Dynamic Fault Trees (DFT) [**?**], Parametric Fault Trees (PFT) [**?**], and Repairable Fault Trees (RFT) [**?**].

State-space models. When the accuracy of combinatorial models is not enough to capture the characteristics of the system to be modelled, state space based models are considered. Models in this category have a greater modelling power and flexibility than combinatorial models, but the state space analysis may be computationally expensive. This depends on the number of states in the model, since the state space size grows exponentially with the number of components in the system. The basic formalism for state-space modelling are *Markovian models*. A Markov process is a stochastic process whose dynamic behaviour is such that probability distributions for its future development depend only on the present state and not on how the process arrived in that state [**?**].

When the state-space is discrete (i.e., the set of all possible values that can be assumed by random variables of the process is discrete), the Markov process is known as a Markov chain. Markov chains are a fundamental block for state-space analysis. They have been used to model a wide number of systems, ranging from network systems, protocols hardware components, software/hardware systems and software applications, complex clustered systems, and for analysing any kind of dependability and performance-related attributes (reliability , availability, performability, survivability). Typical Markovian models are: *discrete time Markov chains* (DTMCs), when the model adopts a discrete index T (usually representing the time), *continuous time Markov chains* (CTMCs), when T is a continuous index, and *Markov Rewards Models* (MRMs), when the Markov chain also includes a reward rate (or weight) attached to the states in order to derive additional measures (e.g., expected accumulated reward in a given interval). The model is usually graphically represented by a state-transition graph, which highlights the system states (the nodes) and transitions among them (the edges) labelled by the one-step transition probability value. When the Markovian memoryless property does not hold (i.e., it does not accurately describe the system being modelled), non-Markovian models (such as *semi-Markov processes* or *Markov regenerative models*), or *non-homogeneous Markov models* are employed, where other distributions are allowed. The accuracy they add to the model is of course paid in terms of complexity in management, parameterization and solution. With the increasing size of systems, this problem led to the introduction of new formalisms and tools. One of this has been particularly successful, due to its ability to concisely represent a complex system in an intuitive fashion: this is *Stochastic Petri Nets* (SPNs). A fundamental feature of SPNs is that there is a direct mapping between them and CTMCs, which allows designers to model their system by the more intuitive SPN formalism, and then to automatically translate it into a CTMC to be solved (by proper tools, such as SPNP, DSPNExpress, GreatSPN, and SHARPE). Similarly, stochastic reward nets (SRN), that are the extension of SPNs with the addition of rewards, can be mapped onto Markov Rewards Models. Motivated by their representational power (their graphical representation is also particularly suited to model parallel architectures, concurrent programs, synchronization problems and multiprocessor systems) and their solution capability as Markovian models, researchers defined several variants of stochastic Petri nets, well-suited to particular application needs or solution methods (*Generalized SPNs, Stochastic Activity Networks (SANs), and Coloured Petri Nets (CPN)*).

Hybrid models: Non-state-space models (e.g., RBDs, FTs) are undoubtedly efficient to specify and analyse, but the independent assumption on which they rely on may be too restrictive for many practical situations. On the other hand, Markovian models provide the ability to model systems that violate this assumption, but at the price of a state space explosion. One way to contain the state-space explosion is hybrid modelling. The main idea is to hierarchically compose/decompose the system and construct models accordingly: state-space methods are used for those parts requiring dependences modelling, whereas combinatorial methods are used for the parts that can be assumed independent.

Several research works try to combine the advantages of combinatorial and state space based analysis methods: typically, minimal subsystems/components are isolated and treated by state-space methods, and then they are combined in a FT-like structure, exploiting combinatorial analysis techniques at the overall system level (these works also fostered proposals to extend the original FT formalism in order to express dependencies by using an FT-like language). Example of works adopting hybrid modelling are in [?] [?] [?] [?] [?] [?] [?]. For instance, authors in [?] modelled a SIP Application Server configuration on WebSphere, by using a set of interacting sub-models of all system components capturing their failure and recovery behaviour. A service reliability analysis adopting a hierarchical model is presented in [?]; the hierarchical modelling is mapped to the physical and logical architecture of the grid service system and makes use of Markov models, Queuing theory, and Graph theory to model and evaluate the grid service reliability.

More generally, the space state explosion problem has triggered many studies based on two general approaches: "largeness avoidance" and "largeness tolerance". Largeness avoidance techniques try to circumvent the generation of large models. They are complemented by largeness tolerance techniques which provide practical modelling support to facilitate the generation and solution of large state-space models. The two main approaches to attack this problem (in which also most of the hybrid models fall) are:

1. compositional approaches, where the system model is constructed in a bottom-up fashion. The models representing parts of the system are built in isolation, and then composed via suitable operators and composition rules;

2. decomposition/aggregation approaches, where the overall model is divided into simpler and more tractable sub-models, and the measures obtained from their solution are then aggregated to compute those concerning the overall model;

Regardless to the modelling formalism adopted, model-based strategies can be distinguished with respect to their objective: *"black box"* models aim to evaluate how the attribute of interest (e.g., reliability) improves during testing and varies after delivery; *"architecture-based models"*, focusing on understanding relationships among system components and their influence on system reliability. Clearly, in this Deliverable we focus on architecture-based modelling, as we are interested in capturing the failing behaviour of a Microservice Architecture. The latter ones can be categorized as: [?]:

- The above-mentioned *State-based* models, that use the control flow graph to represent software architecture; they assume that the transfer of control among components has a Markov property, modeling the architecture as a DTMC, a CTMC, or an SMP or similar.

- *Path-based* models, that consider the possible execution paths of a program.

- *Additive-models*, where the component failure probabilities (or related measures like reliabilities) are modeled by non-homogeneous Poisson process (NHPP) and the system failure intensity is estimated as the sum of the individual components failure intensities.

Failure modelling in MSA. Specific approaches for MSAs are based on Fault Trees [98] or Petri Nets [55] (aiming to reliability assessment), and redundancy of Microservices [55] (aiming to reliability improvement).

Zang *et al.* [98] propose service dependency graph automatic generation scheme and fault tree model. In particular, the service registry is used to obtain the dependencies between microservices. The fault tree model is designed for the system's error tolerance scheme, and the model is quantitatively analyzed according to the execution probability of each execution path.

Zheng *et al.* [55] used Petri nets to model microservice request, microservice, microservice composition, and container, in order to identify the critical microservice in complex cloud applications. In particular, they focus on modeling the reliability of cloud applications based on microservice through a redundancy operation.

Other models are specific for the resiliency provided by MSAs [24, 97]. In particular, in so distributed systems, the performance degradation of a certain system can impact marginally on its availability and reliability. For this reason, the resiliency is preferred as a characteristic describing "the ability to maintain the performance of services at an acceptable level and recover the service back to normal, when a disruption causes the service degradation" [97].

Dullmann *et al.* [24] propose a generative platform for benchmarking performance and resilience engineering approaches in microservice architectures. The approach comprises a metamodel defining the topology of the microservices, a generator for the deployable artifacts of the synthetic microservices, and supporting services for workload generation, problem injection, and monitoring.

4 Architectural modelling

Being able to clearly describe the architecture of the system is an important aspect when trying to reason about it as a whole. To facilitate communication among researchers and practitioners and between technical and non-technical stakeholders, as well as to keep track of the evolution, it is important to have a shared representation of the architecture. This can guide the design of the system, help with exposing and analysing system qualities at different degrees (e.g., performance, maintainability, functional suitability, scalability) and can be seen as basis for understanding the system. At the same time, it can offer a real time visualization of the system by bringing together the results of monitoring. This is especially important in context of MSA-based systems, where the architecture is rapidly evolving and new functionalities are released all the time. To keep up with the fast pace and to deliver in a reliable manner, a common representation of the system is essential. Otherwise, keeping track of tens or hundreds of microservices, with structural and behavioral relations among them can become cumbersome and even impossible at some point.

Of course, the complexity of a software architecture does not fit a one-dimensional model. Different description methods must be considered with respect to the targeted aspects of the system. The so called views are a good way of breaking the multi-dimensions of the architecture into pieces and provide insight into the most important concerns by means of manageable representations. There are 3 main structures which model a system's architecture: *module*, oriented towards the structural, static aspects, *component-and-connector*, behavioral structures which show the runtime behavior and interaction of components and *allocation* structures, highlighting the relation between system components and the environment (hardware, teams etc). [8] We will analyze how these types of views are being used in the context of MSA-based systems.

In the following, we provide a detailed overview of relevant approaches for modeling microservice-based systems. The presented data is based on two systematic mapping studies targeting MSA [22] and MSA in DevOps [33], along with our own literature study of MSA-related publications from specialized venues (ICSA and ECSA).

Approach		Characteris -tics	Description and Uses	References
Box-and -line	Architectural block diagram	- visual - informal notation	Blocks connected by lines are used to represent the relationships between high-level system components. An architectural block diagram provides vision on the basic structure of the system.	[54], [48], [83], [103], [2], [87], [17], [47], [4], [15], [104], [51], [46], [27], [50], [49], [64], [81]
	Functional flow block diagram	- visual - informal notation	Used to represent the functional flow of a system over time. For example, flow diagrams can be used to represent incremental adoption of a DevOps based monitoring solution (OMNIA) [60], or integration of a microservice-based platform (SONATA) with a DevOps methodology (CI/CD) [77].	[60], [41], [29], [6], [77], [100], [15]

	Tiered architecture diagram	- visual - informal notation	Used to separate the MSA into several layers, such as IaaS (Infrastructure as a Service) or infrastructure layer, PaaS (Platform as a Service) or platform layer and SaaS (Software as a Service) or service layer. The IaaS and PaaS layers are used for development, deployment and operation of the software application - the SaaS layer [65], [79]. Similarly, another potential layering of containerized cloud applications can include the hardware layer, the container layer and the software layer [6]. A more coarse-grained architecture separates software services into two tiers - business microservices (bound to business domain) and API microservices (CPU, I/O or memory-intensive microservices, reusable across multiple applications) [100].	[65], [79], [6], [100], [28]

	Flowchart	- visual - informal notation	Used to represent a workflow, or a process, by means of a sequence of steps and decisions. For instance, it can describe the scheduling and execution of containerized microservices in IoT [3], or the incremental integration process	[3], [106]
UML -based	Activity diagrams	- visual - semi-formal notation	Used to present a series of actions, or workflows in a system, such as the migration to microservices and development process [29].	[29], [104]
	Sequence diagrams	- visual - semi-formal notation	Used to present object interactions over time, it can, for instance, show collaboration between the development and integration teams during the development of a microservice-based system (HARNESS), with a DevOps approach [79].	[79]
	Class diagrams	- visual - semi-formal notation	Used for conceptual modeling, it can show the static structure of the tool for incremental integration of microservices [106].	[106]
	Component diagrams	- visual - semi-formal notation - physical model	Used to present the physical model, or the physical components of the system (services, libraries, load balancers etc).	[5], [4]

ADL	TOSCA	- formal notation	OASIS TOSCA (OASIS Topology and Orchestration Specification for Cloud Applications) is the de-facto open standard language for infrastructure-as-code (IasC), based on the "intent modelling" paradigm. It aims to ensure resilient, portable and long-lived orchestrations, by describing the topology of the architecture, with the relationships and dependencies between services hosted on a cloud platform. It uses a Service Template, made of the Topology Template (nodes and their relationships) and Plans, to define the services. Depending on the version, the language is based on XML, or YAML. TOSCA can very well fit in the context of MSA.	[82]

	MicroART-DSL	- formal notation	MicroART is a domain specific language based on EMF (Eclipse Modeling Framework)[10], built around the following concepts: MicroService, ServiceType (functional or infrastructural), Interface, Link (communication among interfaces), Clusters (logical groupings of microservices), Teams and Developers. MicroART, an architecture recovery tool for microservice-based systems conforms to MicroART-DSL. It can generate graphical representations of the MSA, showing the real dependencies between microservices from the development perspective. MicroART-DSL allows representing both the physical and the logical architecture models. These aid architectural analysis and reasoning, provide documentation and validation between the deployed architecture and the designed one and	[37] [40]	

[10]https://www.eclipse.org/modeling/emf/

		help with attributing microservices to their owning teams.	
DIARy	- formal notation	The DIARy-specification-profile is an extension of the SoaML and UML metamodels. It helps create the Extended Increment Architecture Model, used for specifying how an increment architecture will be integrated into an existing cloud services architecture. It requires documenting the architecture of the increment, the logic of the integration and the architectural impact of the integration. Based on DIARy, platform-specific cloud artifacts can be generated, such as source code, environment specific deployment and architectural reconfiguration scripts [106].	[105] [106]

Medley	- formal notation	Medley DSL is a domain-specific language for describing orchestration using high-level constructs and domain-specific semantics, introduced as part of a service decomposition platform with the same name. The Medley specification is focused on business logic - how services are assembled, together with the composition logic, abstracting it from the implementation details.	[9]
Own-DSL	- formal notation	This specification helps representing a microservice environment, by creating an instance of the proposed metamodel [25]. This defines microservice types, dependencies, instances deployed, as well as the API via RESTOperations (URLs are mapped to methods of a microservice) and versioning. Each instance of this metamodel can be used for code artifacts generation (source code, deployment scripts etc).	[25]

	OCCI Extension	- formal notation	It targets the specification of a monitoring infrastructure, provided as a service, in a container-based distributed system. It is based on OCCI (Open Cloud Computing Interface), a standard for describing cloud provisions. The specification includes the core OCCI classes (Resource and Link), together with those in the monitoring extension (Sensor and Collector). Therefore, it allows the definition of a hierarchical monitoring infrastructure, using two types of instances - of measurement and data distribution.	[18]

	UDL	- formal notation	UDL is a XML-based description language for microservices executed on mobile devices. It consists of five sub-languages: - UDL-SP - microservice profile, by providing the descriptive and functional definition of the service - UDL-CD - microservice content description, by defining the received and produced content - UDL-CP - microservice capability profile - UDL-SL - microservice logic, by defining the functional and operational aspects, independent of the platform - UDL-SR - microservice rendering, by defining the abstract, available graphical elements for the service GUI, independent of the platform UDL enables end-user mobile service creation as part of the m:Ciudad platform.	[20]

CAOPLE	- programming language - formal notation	CAOPLE (Caste-centric Agent-Oriented Programming Language and Environment) [96] is an agent-oriented language used for modeling, development, and testing of model-based microservices. In CAOPLE, programs are constructed from castes (caste-centric) and every agent, the basic building block, is an instance of a caste. The CAVM-2 virtual machine provides the runtime of this language. CAOPLE is part of CIDE (CAOPLE Integrated Development Environment), an integrated software development environment built to support continuous testing and seamless integration of microservices [54].	[96], [54], [103]
CAMLE	- formal notation	- CAMLE (Cloud Application Modeling and Execution Language) is a modeling language which enables a tool constructing graphical models of systems. This was integrated into CIDE [54].	[54]

	SLABS	- formal notation	SLABS (Specification Language for Agent-Based Systems) is a model-based formal specification language for agent-based systems integrated into CIDE [54], which can conveniently transform it into CAOPLE. It consists of a set of specifications of agents and castes.	[54] [102]
	Jolie	- formal notation	Jolie[11] is an open-source programming language for developing microservices [61]. The basic building blocks are services that communicate over the network. Jolie is used by JRO (Jolie Redeployment Optimiser), a tool for automatic and optimized deployment of microservice-based systems [34], to support the writing and execution of services.	[34] [61]

[11] https://www.jolie-lang.org/

	SDA	- formal notation	SDA (Service Desiderata Language) is a domain specific language used to express the specification of the target configuration of a microservice-based system. The SDA grammar is capable of defining different constraints, such as the number of desired services, co-location or service distribution.	[34]

Table 20: Architectural languages used with MSA

The inventory gathers different types of views used by researchers to describe, communicate and analyze microservice-based systems. Their representation choices vary from informal box-and-line notation, to UML-based structures and ADLs (architecture description languages) proposed as DSLs (domain-specific languages) for the domain of MSA [57]. The diversity in approaches signals the fact that so far there is no widely adopted and industry-proven standard architecture language for microservices. Depending on the specific requirements of the µDevOps project, we will select the most suitable notation for representing the software architecture of systems in the context of MSA/DevOps.

5 Study on the DSML(s) for µDevOps

In this section we report on our investigation of the set of Domain-Specific Modeling Language(s) (DSMLs) for supporting DevOps in reasoning on the system architecture and coping with the dynamic and changing aspects of the application at runtime. Specifically, we will identify µDevOps-suited behavioural and architectural modelling techniques and DSMLs which will be central to all the other work packages of the project. A first proposal for the modeling techniques and DSML(s) will be presented in the remainder of this section. Feedback about the usage of such modeling techniques and DSML(s) from all project partners involved in WP3, WP4, and WP5 will be collected in order to assess whether an update of the proposed modeling techniques and DSML(s) will be necessary, based on emerging needs.

This study is composed of two main phases: (i) identification of needs and practices of project partners and (ii) proposal for the µDevOps DSML(s).

5.1 Identification of needs and practices of project partners

In order to identify the needs and practices of project partners with respect to the DSML(s) used in µDevOps we designed an online questionnaire and invited all project partners to provide their inputs. The purpose of the questionnaire is to identify and understand the needs, requirements, and practices of both academic researchers and industrial practitioners with respect to the DSML(s) for representing the architecture of microservice-based systems in WP2, WP3, WP4, and WP5 in the µDevOps project.

The target audience of the questionnaire is: *academic researchers and industrial practitioners who will work on WP2, WP3, WP4, and WP5 in the uDEVOPS project.* In the questionnaire, we ask participants to think of their next activities in the context of the µDevOps project they will be involved in, and provide their answers based on those. We designed the questionnaire in such a way that filling it out would take approximately 10 to 15 minutes. The participation to the questionnaire was not anonymous, in order to have the possibility to follow up to the partners in case we need clarifications. In any case, the responses to the questionnaire are handled with care and confidentiality and will never be distributed outside the µDevOps consortium.

We designed the questionnaire based on our experience with previous studies [19, 58], related literature (*e.g.,*), and the results of a recent study we performed on monitoring tools for DevOps and microservice-based systems [36].

The questionnaire is composed of 4 main sections:

- **Expected usage of the DSML in µDevOps**: this section of the questionnaire is about how the participant is planning to use the DSML in the next WPs of the µDevOps project.
- **What needs to be represented**: this section of the questionnaire is about the aspects of the system modelled using the DSML in the next WPs of the µDevOps project.
- **Characteristics of the DSML**: this section of the questionnaire is about the main characteristics of the DSML to be used in the next WPs of the µDevOps project.
- **Closing**: this section wraps up the questionnaire and invites the participant to provide any comments relevant to this study.

Each of the above-listed sections contains a variable number of questions, for a total of 17 individual questions. The specific questions belonging to each section of the questionnaire are described in the remainder of this section, where we will provide the results of the questionnaire.

Overall, the questionnaire was filled out by 10 participants. The set of participants covers four out of six partners of the µDevOps project.

5.1.1 Expected usage of the DSML in µDevOps

The first question is about **whether the participant is already using any DSMLs** for modeling the architecture of microservice-based systems. Participants answered as follows:

- No (6);
- Palladio [71] (2); Palladio is used for architecture documentation and performance and reliability prediction;
- JSON files in µBench [21] (2); µBench is a tool-based approach for the automatic generation (and simulation) of microservice-based systems;
- Swagger[12]/OpenAPI[13] descriptions (2); they are both based on a JSON-based representation of the REST APIs provided by services;
- UML (1) is used for architecture documentation.

When asked about **how the architectural models are going to be used**, the participants provided the following answers:

- For representing monitoring information about the microservices (6);
- For planning/reasoning on the next improvements of the system (5);
- For test cases generation (5);
- For identifying architectural issues within the system (2);
- Used at runtime for self-configuration (0).

Participants **created/updated architectural models** as follows:

- Manually by a developer/engineer (7)
- Automatically in a dynamic way, for example, via monitoring (4);
- Automatically in a static way, for example, via source code analysis (1).

In another question, we asked about **how frequently are the architectural models changing**. Participants answered that architectural models primarily changed *rarely* (4) and *sometimes* (4), followed by *usually* (1) and *don't know* (1). Interestingly, no participant answered that architectural models change *every time*.

When asked about which specific **quality attributes** will be targeted while working with the DSML(s), participants mentioned the following ones:

- Performance (8);

[12] `https://swagger.io`
[13] `https://www.openapis.org`

- Reliability (8);
- Energy efficiency (5);
- Security (3);
- Operability (2);
- Maintainability (2);
- Safety (1);
- Compatibility (0).

Interestingly, the most frequent pairs of considered quality attributes are: performance + reliability (6 answers) and performance + energy efficiency (5 answers). Also, energy efficiency is always paired with performance, highlighting the strict relationship (and involved trade-offs) between those two quality attributes.

The most mentioned **stakeholders using the architectural models** are:

- DevOps engineers (4);
- architects (2);
- testers (2);
- researchers (1).

5.1.2 What needs to be represented

The first question we asked about **what needs to be represented** using the DSML(s) in µDevOps, the participants answered as follows:

- Logical architecture – for example: services dependencies from a development perspective (5);
- Physical architecture – for example: how the services are deployed on physical machines, how they communicate (4);
- Both logical and physical architectures (1).

We also asked participants to provide an indication of which **architectural information needs to be represented in the DSML(s)**. Their answers are reported below:

- Services/microservices (10);
- Communication – who calls whom (9);
- Traces (service calls as they propagate within the system) (8);
- The APIs exposed by each service (7);

- Virtualization aspects, such as containers and/or virtual machines (6);
- Deployment information – for example: hosts, physical machines (5);
- Metrics associated to individual services (5);
- Infrastructural services – for example: service brokers, monitoring services, etc. (3);
- External systems – for example: used external APIs, third-party systems (3);
- Internal behaviour of the services (2);
- Global behaviour of the system (2).

5.1.3 Characteristics of the DSML

In this section we aimed at collecting information about the possible technical characteristics of the DSML(s) to be used in the µDevOps project.

As a starting point, we asked participants to provide any **examples of already-existing DSMLs** that are similar to the one they will need in the µDevOps project. The answers provided by the participants mentioned the following DSMLs: (i) Docker compose[14], (ii) Swagger, (iii) µBench's configuration files, (iv) the Microservice Domain-Specific Language[15] (MDSL), and (v) UML collaboration diagrams augmented with dynamic information (*e.g.,* number of actual calls among microservices). Interestingly, no participant mentioned any language belonging to the Palladio tool suite.

When defining (and using) a DSML it is important to use a concrete syntax that fits well with the modeling ergonomics of the stakeholders using them. So, we asked participants to provide an indication about **which type of architectural representation shall be supported by the DSML**. The answers provided by the participants are reported below:

- visual (8);
- textual (5);
- tabular (3);
- it does not matter since the definition of the DSML's concepts is decoupled from its concrete syntax (1).

When asked about whether the participants would need a **platform-independent representation** of the architecture of the system, the majority of them answered positively (6 participants), followed by 3 participants who did not give any indication, and 1 participant replying negatively.

[14] `https://docs.docker.com/compose`
[15] `https://microservice-api-patterns.github.io/MDSL-Specification/`

In terms of **required modeling technologies** for realizing the DSML(s), the majority of participants did not express any specific requirement (6), followed by an equal number of participants requiring either Eclipse (specifically, EMF) and Jetbrains MPS (2 participants each).

For many years UML has been prominent for modeling the architecture of software-based systems in general [58]. So, we asked participants whether they would prefer to have µDevOps DSML(s) **based on UML**. Their answers are generally positive, with the following answers:

- Strongly agree (2);
- Agree (5);
- Neutral (0);
- Disagree (1);
- Strongly disagree (0);
- Don't know (2).

Finally, when asked which **level of formality** shall have the description of the semantics of the DSML(s) in µDevOps, the participants answered as follows:

- Semi-formal – a la UML (8);
- Formal – each concept of the DSML is specified mathematically (1);
- Informal – textual description of the main concepts of the DSML (0);
- Don't know (1).

* * *

In the **closing** section of the questionnaire, we received two comments. The first one is about one of the partners giving their availability to provide more information about the Palladio component model (with a link to a downloadable executable of its supporting modeling tool). The second comment nicely summarizes the expectations of the partners; we report such comment verbatim: "*Concepts in the remaining WPs are about testing/quality attributes (WP3), risk assessment (WP4) which will mostly mix the quality attributes with usage profiles to assess a risk, and deployment on Cloud of the µDevOps prototype. Any concept related to this could be useful in a DSML*".

5.2 Proposal for the µDevOps DSML(s)

By analysing the results of the online questionnaire discussed in the previous section, it is interesting to note the **heterogeneity of the requirements** of the µDevOps consortium. For example, when asked about the concepts that should

be represented in the DSML, participants highlighted the need for representing services and microservices, how the services communicate (*i.e.,*, the service mesh), call traces, the APIs exposed by the services, deployment information, and even metrics associated to services.

Overall, we extracted the following recurrent needs and practices, which are relevant for the DSML(s) to be used in µDevOps. First of all, the DSML(s) will be used for **forward engineering**, but also for **monitoring** and **test cases generation**; in the first case, the architectural models are manually created and edited by a developer/engineer, whereas in the second case they are dynamically (and automatically) extracted from the running system via monitoring. Also, participants did not highlight a general need to have an always up-to-date representation of the architecture of the system; this means that there is **no need to integrate the DSML with a dedicated highly-performant model extractor**. According to the general interests within the µDevOps project consortium and goals, the most targeted quality attributes are **performance**, **reliability**, and **energy efficiency** of microservice-based system, with a primary focus on their **logical and physical architectures**.

In terms of technical realization of the DSML(s), participants indicated to need mainly a **visual and textual representation of the architecture** (at best, integrated with each other). Also, the majority of participants indicated the need to have a **platform-independent representation** of the architecture of the system, with **support for a UML-based representation**, which is by definition semi-formal. Finally, some participants highlighted the need to realize the DSML(s) based on **different modeling platforms**, such as the Eclipse Modeling Framework (EMF) and Jetbrains MPS.

Based on the requirements elicited above, having a single DSML supporting all needs and practices of µDevOps partners will be counterproductive for the project. Indeed, such a single DSML will inevitably contain a large set of potentially unrelated concepts, belonging to different levels of abstraction. All concepts of such a hypothetical large language will need to be defined from scratch, maintained, and their consistency will need to be ensured (at least semi-formally). Moreover, the modeling tool for such a unique language will need to be realized from scratch and across multiple modeling platforms, basically duplicating years of effort spent by multiple research and development teams on already-existing modeling languages for the architecture of microservice-based systems.

In the context of µDevOps, we propose to follow the principle of "using the right tool for the right job" and invite project partners to **reuse already-existing modeling languages** in the project, where each modeling language will support at best their needs and practices in the specific work packages of the project. Based on the information collected in the online survey, project partners might use: (i) Palladio, (ii) the configuration files of µBench, (iii) Open API specifications, and (iv) UML.

We are aware that using multiple DSMLs in parallel might raise the risk of having inconsistencies (both syntactical and semantic) among models representing the same system [59]. If some level of portability of the information

belonging to multiple models is necessary, we suggest developing **model-to-model transformations** to automatically transform models conforming to a DSML (*e.g.,* Palladio) to models conforming to other DSMLs (*e.g.,* Open API specifications or µBench configuration files). Thanks to the conceptual alignment between Palladio and UML, some steps towards this direction have been already made in the context of Palladio, for which a model-to-model transformation to UML is already existing[16]; model-to-model transformations from UML to the Palladio component model are already existing as well [12].

Finally, we would like to draw the attention of project partners to the MicroArt DSML [38, 39]. It is a platform-independent DSML for representing the architecture of microservice-based systems. The MicroArt DSML has been designed by VU researchers around general microservice needs and characteristics and it is kept minimal in order to support the design and description of multiple microservice-based systems in a simple, but effective manner. The main concepts of the MicroArt DSML include: product, microservice, interface (with endpoints), communication links, host, cluster, developer, development team, and various service types. The MicroArt DSML also allows DevOps engineers to represent and evaluate software quality attributes for the architecture of microservice-based systems [14]. Being it minimal and tailored to microservices, the MicroArt DSML might be used by project partners as the *lingua franca* within the µDevOps project consortium, both for communication and more technical purposes.

6 Conclusion

The deliverable reported about the activities carried out in work package 2. This pertained to the main aspects for characterizing a Microservice Development Operations Engineering context, for what we called "context modelling" aimed at context-driven quality assurance. We surveyed the different ways of gathering information through monitoring; these are a crucial aspect in microservice archietctures, as allow to gather data then used for parameterizing decision-support models. An extensive survey has been conducted about which monitoring tools are available and what are their features (what they monitor, how they monitor). The second part was about modelling: what are the formalisms that can be exploited to model the architecture and the correct and failing behaviour of a system, possibly with the support of a DSML. The resulting models allow supporting a plethora of quality-related activities, ranging from testing, to root cause analysis, to KPI prediction. The results are being used in the related WPs 3 and 4 and will be used in the final WP 5.

[16] `https://github.com/PalladioSimulator/Palladio-Addons-PlantUML`

References

[1] Alberto Avritzer, Daniel Menasché, Vilc Rufino, Barbara Russo, Andrea Janes, Vincenzo Ferme, André van Hoorn, and Henning Schulz. Pptam: Production and performance testing based application monitoring. In *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 39–40, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Jeongju Bae, Chorwon Kim, and JongWon Kim. Automated deployment of smartx iot-cloud services based on continuous integration. In *2016 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1076–1081, 2016.

[3] Jeongju Bae, Chorwon Kim, and Jongwon Kim. Automated deployment of smartx iot-cloud services based on continuous integration. pages 1076–1081, 10 2016.

[4] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to cloud-native architectures using microservices: An experience report. pages 201–215, 07 2015.

[5] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[6] Cornel Barna, Hamzeh Khazaei, Marios Fokaefs, and Marin Litoiu. Delivering elastic containerized cloud applications to enable devops. 05 2017.

[7] L. Bass, I.M. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Always learning. Addison-Wesley, 2015.

[8] Len Bass and Rick Kazman. *Software Architecture In Practice*. 01 2003.

[9] Elyas Ben Hadj Yahia, L. Reveillere, Bromberg Yerom, Raphaël Chevalier, and Alain Cadot. Medley: An event-driven lightweight platform for service composition. volume 9671, pages 3–20, 06 2016.

[10] Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardoso. Automated analysis of distributed tracing: Challenges and research directions. *Journal of Grid Computing*, 19:1–15, 2021.

[11] Antonia Bertolino, Guglielmo De Angelis, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. Devopret: Continuous reliability testing in devops. *Journal of Software: Evolution and Process*, n/a(n/a):e2298, 2020. e2298 smr.2298.

[12] Andreas Brunnert, Alexandru Danciu, Christian Vögele, Daniel Tertilt, and Helmut Krcmar. Integrating the palladio-bench into the software development process of a soa project. In *KPDAYS*, pages 30–38, 2013.

[13] Matteo Camilli, Antonio Guerriero, Andrea Janes, Barbara Russo, and Stefano Russo. Microservices integrated performance and reliability testing. In *2022 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2022.

[14] Mario Cardarelli, Ludovico Iovino, Paolo Di Francesco, Amleto Di Salle, Ivano Malavolta, and Patricia Lago. An extensible data-driven approach for evaluating the quality of microservice architectures. In *Proceedings of the 34th Annual ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 08-12, 2019*, pages 1225–1234, 2019.

[15] Cloves Carneiro and Tim Schmelmer. *Microservices from day one: : Build Robust and Scalable Software from the Start*, pages 151–174. Springer, 2016.

[16] N. Carstensen. What is log management? a complete logging guide., 2020.

[17] Pethuru Raj Chelliah and Anupama Raman. *Automated Multi-cloud Operations and Container Orchestration*, pages 185–218. 05 2018.

[18] Augusto Ciuffoletti. Automated deployment of a microservice-based monitoring infrastructure. *Procedia Computer Science*, 68:163–172, 12 2015.

[19] Istvan David, Kousar Aslam, Ivano Malavolta, and Patricia Lago. Collaborative model-driven software engineering – a systematic survey of practices and needs in industry. *Journal of Systems and Software*, 199:111626, 2023.

[20] Marcin Davies, François Carrez, Juhani Heinilä, Anna Fensel, Maribel Narganes, and José Carlos dos Santos Danado. m:ciudad: enabling end-user mobile service creation. *International Journal of Pervasive Computing and Communications*, 7(4):384–414, 2011.

[21] Andrea Detti, Ludovico Funari, and Luca Petrucci. μbench: an open-source factory of benchmark microservice applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(3):968–980, 2023.

[22] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150:77–97, 2019.

[23] Salvatore Distefano and Antonio Puliafito. Dependability evaluation with dynamic reliability block diagrams and dynamic fault trees. *IEEE Transactions on Dependable and Secure Computing*, 6(1):4–17, 2009.

[24] Thomas F. Düllmann and André van Hoorn. Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, page 171–172, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Thomas F. Düllmann and André van Hoorn. Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, page 171–172, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *IEEE Software*, 33(3):94–100, 2016.

[27] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *IEEE Software*, 33(3):94–100, 2016.

[28] Bob Familiar. *What is a Microservice? Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions, Chapter 2.* Springer, 2015.

[29] Chen-Yuan Fan and Shang-Pin Ma. Migrating monolithic mobile application to microservice architecture: An experiment report. In *2017 IEEE International Conference on AI Mobile Services (AIMS)*, pages 109–112, 2017.

[30] Maximiliano Firtman. *Hacking Web Performance.* O'Reilly Media, Inc., Sebastopol, CA, USA, June 2018.

[31] J. L. Fleiss, B. Levin, and M. C. Paik. *The Measurement of Interrater Agreement*, chapter 18, pages 598–626. John Wiley & Sons, Ltd, 2003.

[32] Susan J Fowler. *Production-ready microservices: building standardized systems across an engineering organization.* " O'Reilly Media, Inc.", 2016.

[33] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 21–30, 2017.

[34] Maurizio Gabbrielli, Saverio Giallorenzo, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. *Self-Reconfiguring Microservices*, volume 9660, pages 194–210. 03 2016.

[35] Javad Ghofrani and Daniel Lübke. Challenges of microservices architecture: A survey on the state of the practice. volume 2072, pages 1 – 8, 2018.

[36] Luca Giamattei, Antonio Guerriero, Roberto Pietrantuono, Stefano Russo, Ivano Malavolta, Tanjina Islam, Madalina Dînga, Anne Koziolek, Snigdha Singh, Martin Armbruster, José-María Gutierrez-Martinez, Sergio Caro-Alvaro, Daniel Rodriguez, Sebastian Weber, Jorg Henss, Estrella Fernandez Vogelin, and Fernando Simon Panojo. Monitoring tools for devops and microservices: A systematic grey literature review. *Journal of Systems and Software*, 208:111906, 2024.

[37] Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. Microart: A software architecture recovery tool for maintaining microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 298–302, 2017.

[38] Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. Microart: A software architecture recovery tool for maintaining microservice-based systems. In *Proceedings of the 14th International Conference on Software Architecture (ICSA)*, pages 298–302. IEEE, 2017.

[39] Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. Towards recovering the software architecture of microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017*, pages 46–53, April 2017.

[40] Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. Towards recovering the software architecture of microservice-based systems. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 46–53, 2017.

[41] Jihun Ha, Jungyong Kim, Heewon Park, Jaehong Lee, Hyuna Jo, Heejung Kim, and Jaeheon Jang. A web-based service deployment method to edge devices in smart factory exploiting docker. In *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 708–710, 2017.

[42] Stefan Haselböck, Rainer Weinreich, and Georg Buchgeher. An expert interview study on areas of microservice design. In *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*, pages 137–144, 2018.

[43] Josune Hernantes, Gorka Gallardo, and Nicolas Serrano. It infrastructure-monitoring tools. *IEEE software*, 32(4):88–93, 2015.

[44] Darby Huye, Yuri Shkuro, and Raja R Sambasivan. Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432. USENIX Association, 2023.

[45] Ramtin Jabbari, Nauman bin Ali, Kai Petersen, and Binish Tanveer. What is devops? a systematic mapping study on definitions and practices. In *Proceedings of the Scientific Workshop Proceedings of XP2016*, XP '16 Workshops, New York, NY, USA, 2016. Association for Computing Machinery.

[46] Kai Jander, Lars Braubach, and Alexander Pokahr. Defense-in-depth and role authentication for microservice systems. *Procedia Computer Science*, 130:456–463, 01 2018.

[47] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. *Challenges When Moving from Monolith to Microservice Architecture*, pages 32–47. 02 2018.

[48] Hui Kang, Michael Le, and Shu Tao. Container and microservice driven design for cloud infrastructure devops. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 202–211, 2016.

[49] Holger Karl, Sevil Dräxler, Manuel Peuster, Alex Galis, Michael Bredel, Aurora Ramos, Josep Martrat, Muhammad Shuaib Siddiqui, Steven van Rossem, and Wouter et al. Tavernier. Devops for network function virtualisation: an architectural approach. *Transactions on Emerging Telecommunications Technologies*, 27(9):1206–1215, 2016.

[50] Stefan Kehrer and Wolfgang Blochinger. Autogenic: Automated generation of self-configuring microservices. In *CLOSER*, 2018.

[51] Chorwon Kim, Seungryong Kim, and Jongwon Kim. *Understanding Automated Continuous Integration for Containerized Smart Energy IoT-Cloud Service*, pages 1275–1280. 01 2018.

[52] B. Kitchenham and P. Brereton. A systematic review of systematic review process research in software engineering. *Inf. Softw. Technol.*, 55(12):2049–2075, dec 2013.

[53] H. Knoche and W. Hasselbring. Drivers and barriers for microservice adoption – a survey among professionals in germany. *Enterpr. Model. Inf. Syst. Archit. - Int. J. Concept. Model.*, page 1–35, 2019.

[54] Desheng Liu, Hong Zhu, Chengzhi Xu, Ian Bayley, David Lightfoot, Mark Green, and Peter Marshall. Cide: An integrated development environment for microservices. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 808–812, 2016.

[55] Zheng Liu, Guisheng Fan, Huiqun Yu, Liqiong Chen, and Xiaoxian Yang. An approach to modeling and analyzing reliability for microservice-oriented cloud applications. 2021, January 2021.

[56] Michael R. Lyu. Software reliability engineering: A roadmap. In *Future of Software Engineering (FOSE)*, pages 153–170. IEEE, 2007.

[57] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, 39, 12 2012.

[58] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, June 2013.

[59] Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Damien Tamburri. Providing architectural languages and tools interoperability through model transformation technologies. *IEEE Transactions on Software Engineering*, 36(1):119–140, jan 2010.

[60] Marco Miglierina and Damian Tamburri. Towards omnia: A monitoring factory for quality-aware devops. pages 145–150, 04 2017.

[61] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Jolie: a java orchestration language interpreter engine. *Electronic Notes in Theoretical Computer Science*, 181:19–33, 06 2007.

[62] John D. Musa and William W. Everett. Software-reliability engineering: Technology for the 1990s. *IEEE Software*, 7(6):36–43, November 1990.

[63] D.M. Nicol, W.H. Sanders, and K.S. Trivedi. Model-based evaluation: from dependability to security. *IEEE Transactions on Dependable and Secure Computing*, 1(1):48–65, 2004.

[64] Rory O'Connor, Peter Elger, and Paul Clarke. Continuous software engineering—a microservices architecture perspective. *Journal of Software: Evolution and Process*, 29, 04 2017.

[65] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. Architectural principles for cloud software. *ACM Transactions on Internet Technology*, 18, 06 2017.

[66] K. Petersen, S. Vakkalanka, and L. Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, 2015.

[67] R. Pietrantuono, S. Russo, and A. Guerriero. Run-time Reliability Estimation of Microservice Architectures. In *IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 25–35. IEEE, 2018.

[68] Roberto Pietrantuono, Antonia Bertolino, Guglielmo De Angelis, Breno Miranda, and Stefano Russo. Towards continuous software reliability testing in devops. In *Proceedings of the 14th International Workshop on Automation of Software Test*, pages 21–27. IEEE, 2019.

[69] Roberto Pietrantuono, Stefano Russo, and Antonio Guerriero. Testing microservice architectures for operational reliability. *Software Testing Verification and Reliability*, 30(2), 2020.

[70] Dinesh Rajput. *Hands-On Microservices–Monitoring and Testing: A performance engineer's guide to the continuous testing and monitoring of microservices*. Packt Publishing Ltd, 2018.

[71] Ralf Reussner, Steffen Becker, Erik Burger, Jens Happe, Michael Hauck, Anne Koziolek, Heiko Koziolek, Klaus Krogmann, and Michael Kuperberg. The palladio component model. 2011.

[72] C. Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018.

[73] C. Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018.

[74] Jasper Van Riet, Ivano Malavolta, and Taher Ahmed Ghaleb. Optimise along the way: An industrial case study on web performance. *Journal of Systems and Software*, 198:111593, 2023.

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

[76] Yuri Shkuro. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing Ltd, 2019.

[77] Thomas Soenen, Steven Rossem, Wouter Tavernier, Felipe Vicens, Dario Valocchi, Panos Trakadas, Panos Karkazis, George Xilouris, Philip Eardley, Stavros Kolometsos, Michail Kourtis, Daniel Guija, Muhammad Shuaib Siddiqui, Peer Hasselmeyer, Jose Bonnet, and Diego Lopez. Insights from sonata: Implementing and integrating a microservice-based nfv service platform with a devops methodology. pages 1–6, 04 2018.

[78] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[79] Mark Stillwell and Jose G. F. Coutinho. A devops approach to integration of software components in an eu research project. QUDOS 2015, page 1–6, New York, NY, USA, 2015. Association for Computing Machinery.

[80] D. Swersky. The hows, whys and whats of monitoring microservices., 2020.

[81] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

[82] Damian Tamburri, Willem-Jan Heuvel, Chris Lauwers, Paul Lipton, Derek Palma, and Matt Rutkowski. Tosca-based intent modelling: goal-modelling for infrastructure-as-code. *SICS Software-Intensive Cyber-Physical Systems*, 34, 06 2019.

[83] Tran Quang Thanh, Stefan Covaci, Thomas Magedanz, Panagiotis Gouvas, and Anastasios Zafeiropoulos. Embedding security and privacy into the development and operation of cloud applications and services. In *2016 17th International Telecommunications Network Strategy and Planning Symposium (Networks)*, pages 31–36, 2016.

[84] TIOBE. TIOBE Index, June 2022. [Online; accessed 17. Jul. 2023].

[85] Paolo Tonella and Filippo Ricca. Statistical testing of web applications. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(1-2):103–127, 2004.

[86] C. Trammell. Quantifying the reliability of software: statistical testing based on a usage model. In *Proceedings of Software Engineering Standards Symposium*, pages 208–218, 1995.

[87] Demetris Trihinas, Athanasios Tryfonos, Marios D. Dikaiakos, and George Pallis. Devops as a service: Pushing the boundaries of microservice adoption. *IEEE Internet Computing*, 22(3):65–71, 2018.

[88] Roberto Verdecchia, Patricia Lago, Christof Ebert, and Carol De Vries. Green it and green software. *IEEE Software*, 38(6):7–15, 2021.

[89] Markos Viggiato, Ricardo Terra, Henrique Rocha, Marco Túlio Valente, and Eduardo Figueiredo. Microservices in practice: A survey study. *ArXiv*, abs/1808.04836, 2018.

[90] Manish Virmani. Understanding devops amp; bridging the gap from continuous integration to continuous delivery. In *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, pages 78–82, 2015.

[91] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, and H. Krcmar. WESSBAS: Extraction of probabilistic workload specifications for load testing and performance prediction–a model-driven approach for session-based application systems. *Software & Systems Modeling*, 17(2):443–477, 2018.

[92] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering*, 26(4):63, 2021.

[93] Muhammad Waseem, Peng Liang, and Mojtaba Shahin. A systematic mapping study on microservices architecture in devops. *Journal of Systems and Software*, 170:110798, 2020.

[94] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, 182:111061, 2021.

[95] J.A. Whittaker and M.G. Thomason. A markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, 1994.

[96] Chengzhi Xu, Hong Zhu, Ian Bayley, David Lightfoot, Mark Green, and Peter Marshall. Caople: A programming language for microservices saas. In *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 34–43, 2016.

[97] Kanglin Yin and Qingfeng Du. On representing resilience requirements of microservice architecture systems. *International Journal of Software Engineering and Knowledge Engineering*, 31(06):863–888, 2021.

[98] Zhigang Zang, Qiaoyan Wen, and Kangming Xu. A fault tree based microservice reliability evaluation model. *IOP Conference Series: Materials Science and Engineering*, 569:032069, 08 2019.

[99] He Zhang, Shanshan Li, Zijia Jia, Chenxing Zhong, and Cheng Zhang. Microservice architecture in reality: An industrial inquiry. In *2019 IEEE international conference on software architecture (ICSA)*, pages 51–60. IEEE, 2019.

[100] Tianlei Zheng, Xi Zheng, Yuqun Zhang, Yao Deng, ErXi Dong, Rui Zhang, and Xiao Liu. Smartvm: a sla-aware microservice deployment framework. *World Wide Web*, 22, 01 2019.

[101] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2021.

[102] Hong Zhu. Slabs: A formal specification language for agent-based systems. *International Journal of Software Engineering and Knowledge Engineering*, 11:529–558, 10 2001.

[103] Hong Zhu and Ian Bayley. If docker is the answer, what is the question? In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 152–163, 2018.

[104] Sergey Zykov. *Agile Services*, pages 65–105. 04 2018.

[105] Miguel Zúñiga Prieto, Emilio Insfran, and Silvia Abrahão. Architecture description language for incremental integration of cloud services architectures. 10 2016.

[106] Miguel Zúñiga Prieto, Emilio Insfran, Silvia Abrahão, and Carlos Cano Genoves. *Automation of the Incremental Integration of Microservices Architectures*, pages 51–68. 04 2017.