Empirical Software Engineering (2025) 30:128
https://doi.org/10.1007/s10664-025-10686-2

Comparison of static analysis architecture recovery tools for microservice applications

Simon Schneider[1] · **Alexander Bakhtin**[2] · **Xiaozhou Li**[3] · **Jacopo Soldani**[4] · **Antonio Brogi**[4] · **Tomas Cerny**[5] · **Riccardo Scandariato**[1] · **Davide Taibi**[2]

Accepted: 3 June 2025

Abstract
Architecture recovery tools help software engineers obtain an overview of the structure of their software systems during all phases of the software development life cycle. This is especially important for microservice applications because they consist of multiple interacting microservices, which makes it more challenging to oversee the architecture. Various tools and techniques for architecture recovery (also called architecture reconstruction) have been presented in academic and gray literature sources, but no overview and comparison of their accuracy exists. This paper presents the results of a multivocal literature review with the goal of identifying architecture recovery tools for microservice applications and a comparison of the identified tools' architectural recovery accuracy. We focused on static tools since they can be integrated into fast-paced CI/CD pipelines. 13 such tools were identified from the literature and nine of them could be executed and compared on their capability of detecting different system characteristics. The best-performing tool exhibited an overall F1-score of 0.86. Additionally, the possibility of combining multiple tools to increase the recovery correctness was investigated, yielding a combination of four individual tools that achieves an F1-score of 0.91.

Keywords Microservices · Architecture recovery · Architecture reconstruction · Static analysis

Communicated by: Gema Rodriguez-Perez and Ben Hermann.

Registered report: The methodology of this study has been peer-reviewed and accepted as a registered report at MSR'24: https://arxiv.org/abs/2403.06941

Extended author information available on the last page of the article

Published online: 20 June 2025

1 Introduction

Static analysis tools can support developers with valuable feedback on their work without the need to run and test their systems. Tools such as SonarQube[1], PMD[2], or IntelliJ[3] are examples of widely popular solutions that perform real-time analyses for different aspects of development. Architecture recovery tools (also called *architecture reconstruction*) can support developers by showing the implemented system's high-level architectural design, helping them adhere to the intended design and avoid architectural issues such as violations of security rules (Bambhore Tukaram et al. 2022), bad smells (Ponce et al. 2022), antipatterns (Taibi et al. 2020), or non-conformances (Cao et al. 2024). Providing accessibility of the systems' architecture is additionally important for microservice systems, consisting of tens, often hundreds, of interacting microservices.

The microservice architecture is an architectural style that often entails a distributed codebase. It has been increasingly adopted in the last years and continues to gain popularity in software development. Applications employing the microservice architecture split their business logic into multiple microservices. The individual microservices communicate over lightweight communication channels (Dragoni et al. 2013; Lewis and Fowler 2014). The architecture has many benefits for software engineering activities. However, the distributed nature of the codebase can pose challenges, since it is more difficult to gain and maintain an overview of the application's architectural design (Di Francesco et al. 2019; Soldani et al. 2018).

Architecture recovery tools and techniques can support developers and software architects in this regard by creating a representation of the implemented system. In the field of program comprehension, it has been shown that such representations foster better and easier analysis, maintainability, and usability, and support software engineers during development (e.g., Arisholm et al. 2006; Budgen et al. 2011; Gravino et al. 2010, 2015; Schneider et al. 2024). By allowing to assess the adherence of the implemented architecture to the designed one, issues such as architectural drift are also mitigated.

With the growing adoption of the microservice architecture, the need for static analysis tools that specialize in microservice applications rises. Consequently, various approaches for architecture recovery for microservices have been proposed in the academic literature (Alshuqayran et al. 2018; Granchelli et al. 2017; Kleehaus et al. 2018; Quéval and Zdun 2023; Soldani et al. 2021). Some authors also provide tools to show the feasibility of their presented approaches. Further tools can be found in the gray literature on the topic.

Among these tools, those that follow a static approach (as opposed to dynamic or hybrid approaches) are especially attractive for use in modern software engineering practices, where rapid development cycles are the norm. In such fast-paced scenarios, lightweight techniques are preferred since they can be better plugged into CI/CD (continuous integration/continuous delivery) pipelines without the need for complex analysis environments and without impeding timely processing.

In this paper, we present a study that we conducted to identify and compare static analysis tools for architecture recovery of microservice applications. A core contribution is the

[1] https://www.sonarsource.com/products/sonarqube/

[2] https://pmd.github.io/

[3] https://www.jetbrains.com/de-de/idea/

execution of all identified tools on a common dataset and the comparison of their accuracy in performing architecture recovery. We measure the extraction correctness in terms of precision, recall, and F1-score of extracted application characteristics, compared to a manually created ground truth.

In the context of this work, we refer to a microservice application's architecture as a representation of its architectural design in terms of components and their logical connections. The microservice architecture offers such a system decomposition by definition, since individual microservices are meant to be self-standing, independently deployable units. Communication links between them that are necessary to fulfill the system's business logic form the connections between components.

Research Questions In pursuing to fulfill the above objectives, this paper addresses the following research questions:

- **RQ1: Which freely available, static analysis tools for architecture recovery of microservice applications exist?** Various architecture recovery approaches have been proposed in the literature, which are often supported by prototypes implementing the techniques. Additionally, gray literature sources give pointers to further tools that fit this scope. We curated a list of such tools in a multivocal literature review and evaluated and compared them on a common benchmark.
- **RQ2: Which characteristics do the tools extract in addition to the basic architecture (i.e., components and connections between them)?** This work is mainly concerned with the reconstruction of the analyzed application's basic architecture, i.e., services and connections between them. However, many tools extract additional application *characteristics*, such as information about implemented security mechanisms, links to design requirements, or trust boundaries. An overview of these additional characteristics helps to identify tools for specific use cases.
- **RQ3: Which are the most commonly considered characteristics extracted by the tools?** Based on the presentation of the characteristics extracted by the identified tools, we analyzed the tools' overlaps and differences in their extraction scopes. The results show what the tools mostly focus on and also where gaps lie.
- **RQ4: Which is the identified tools' accuracy in architecture recovery?** To compare the identified tools based on their correctness in architecture recovery, we executed them on a common benchmark and measured their precision, recall, and F1-score concerning the following properties:
  - **RQ4.1: Which is the identified tools' accuracy in detecting the components that form a microservice application?** The individual microservices of an application constitute the building blocks that the application's microservice architecture consists of. The components are often the easiest characteristics to extract for microservice applications, since deployment technologies such as Docker Compose or Kubernetes ease their detection. Nevertheless, this is the foundational step of architecture recovery and was thus evaluated.
  - **RQ4.2: Which is the identified tools' accuracy in detecting connections between the components forming a microservice application?** The second characteristic in the core architecture of microservice applications is the connections between the

components, over which requests are made and data is exchanged. Such connections can be realized in different ways, for example via direct API calls, asynchronous communication techniques, or implicit invocations by infrastructural components such as the communication for registering services in a service registry. Due to this added complexity, the detection of connections is harder than that of the components and is evaluated separately.

- **RQ4.3: Which is the tools' accuracy in detecting the additionally extracted characteristics?** For those tools that extract characteristics in addition to the basic architecture (see RQ2), we measured their correctness in extracting this extra information. We compared tools that extract the same additional characteristics.

- **RQ5: Can combinations of multiple tools outperform the best individual tools?** Combinations of multiple individual tools could show synergies that improve the results over those observed when executing the tools alone. We investigated multiple combinations of tools concerning the possibility of improving the architecture recovery correctness in all three evaluated metrics (precision, recall, and F1-score).

This paper provides an overview and comparison of state-of-the-art, freely available, static analysis architecture recovery tools for microservice applications. The presented findings can be beneficial to both researchers and practitioners. Insights on the quality of such tools, as well as on the specific characteristics they extract, are valuable for both groups of stakeholders. For researchers, an overview of existing tools can prevent the creation of yet another approach for which a similar technique has already been proposed. Consequently, the results of the study can shed light on the directions for future work and help accelerate research on the topic. For practitioners, the results of the study can provide a reference for the tools and for comparing their actual capabilities. Industry adoption of tools and techniques presented in academic literature is notoriously challenging. Our study is geared towards fostering visibility of tools for microservice architecture recovery and showing their accuracy. With the comparison based on results observed from executing all tools on the same applications instead of an overview of their approaches, our study fills a gap in the literature that has not been addressed before, to the best of our knowledge. Especially in academia, where published tools are often prototypes created for the sake of showing the feasibility of a presented approach and where subsequent maintenance is often neglected, such an evaluation is crucial for properly judging the tools' qualities.

The rest of this paper is structured as follows: Section 2 provides an extended and updated description of the methodology presented in the registered report of this work. Section 3 describes the MLR and the tools identified with it. Section 4 presents the comparison of the identified tools concerning their extraction correctness and Section 5 discusses these results and presents lessons learned. Section 6 describes limitations of the presented work. Finally, Section 7 presents the related work and Section 8 concludes the paper.

2 Methodology

The methodology of the conducted study has been presented as a registered report at the International Conference on Mining Software Repositories 2024 (MSR'24) (Schneider et al. 2024). We repeat it below in updated form for consistency of this paper. Research question RQ5 and the related methodology (presented in Section 2.6) is an extension with respect to the registered report.

The presented work consisted of two parts, (i) a multivocal literature review to identify static analysis architecture recovery tools for microservice applications, and (ii) a comparison of the identified tools' accuracy in architecture recovery by executing them on a common dataset and evaluating the outputs they produce. Figure 1 shows the complete methodology structured into five steps. Each step is further described in the following sections, in the order indicated by the figure.

2.1 Identification of tools

The systematic literature review in this work is a replication of the one presented by Bakhtin et al. (2024). The search has been repeated to identify tools published after the authors performed their search. Further, the in- and exclusion criteria of our study have been applied to the tools already identified in the replicated literature review to select those relevant to us. The original list is not restricted to a specific analysis approach, while we focus on static analysis tools, i.e., a subset of the original list. Because of this and since the methodology we applied is adapted from the one of Bakhtin et al. (2024), no relevant tools were missed in this process. Additionally to academic sources, we adopted the methodology and applied it to gray literature sources as well, thus extending the work into a multivocal literature review (Garousi et al. 2019) (see step ① in Fig. 1).

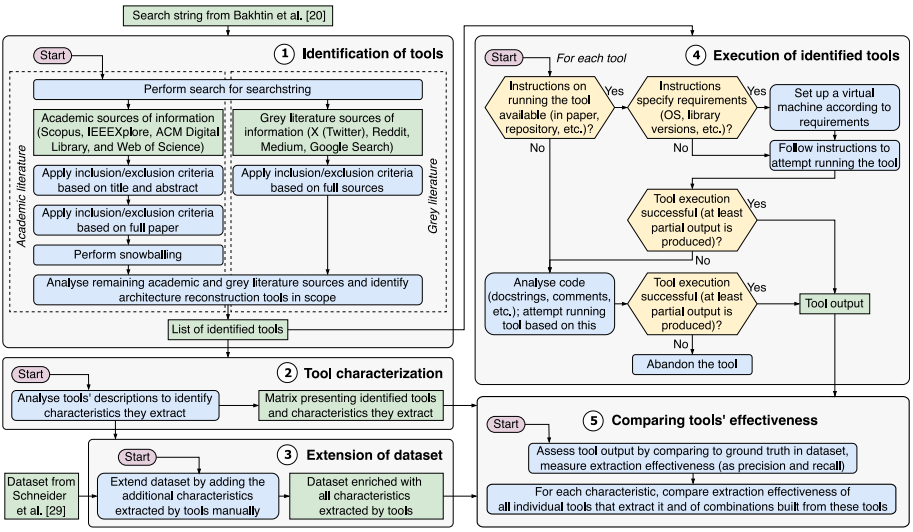

Fig. 1 The methodology adopted in this study

For the repetition of the literature review of Bakhtin et al. (2024), we used the same search string (given below) and searched for it in the same four scientific databases (Scopus,[4], IEEEXplore[5], ACM Digital Library,[6] and Web of Science[7]). We only considered results published after the search date reported in the paper of the original literature review.

Search string:

```
(Microservice* OR Micro-service* OR "micro-service*")
AND Architect*
AND (Reconstr* OR Mining OR Reverse engineering
OR Recover* OR Extract* OR Discover*)
AND (Tool* OR Prototype OR Implementation OR GitHub OR
Proof of concept OR POC OR Proof-of-concept)
```

The fourth AND-group of the search string is used for in-text search if the database functionality allows it.

For gray literature sources, we applied the search string to four websites: Google Search[8], X (Twitter)[9], Reddit[10], and Medium[11]. These are popular websites for a technical audience and have been used as sources of gray literature resources in the literature (Moreschini et al. 2023; Peltonen et al. 2021). Duplicates were removed during result aggregation. After compiling the list of initial sources, we applied the following inclusion and exclusion criteria (adapted from Bakhtin et al. (2024)), to fit our target scope:

Inclusion criteria:

1. Mentions a tool for microservice architecture recovery
2. A reference to the freely available tool is made or the tool can be found by searching for its name
3. The tool follows a static or hybrid analysis approach**Exclusion criteria:**

1. Source is not in English
2. Out of topic – relevant terms are used in a different context
3. Source describes different aspects of microservice recovery (not dealing with tools)
4. Source describes closely related tasks such as monolith to microservice migration

We applied the criteria in two phases, as it is common practice when conducting SLRs (Kitchenham 2004; Kitchenham and Charters 2007; Ralph et al. 2020). For the academic sources, sources were excluded based on reading the title and abstract of the paper in the first phase; in the second phase, the complete paper was examined. For the gray literature sources, we considered the title in the first phase and the full content including comments in the second phase. In each phase, sources were excluded that met any of the formulated exclusion cri-

[4] Scopus: https://www.scopus.com.

[5] IEEEXplore: https://ieeexplore.ieee.org/.

[6] ACM Digital Library: https://dl.acm.org.

[7] Web of Science: https://www.webofscience.com/wos/woscc/basic-search

[8] Google Search: https://google.com

[9] X (Twitter): https://x.com

[10] Reddit: https://reddit.com

[11] Medium: https://medium.com

teria or if it could be decided that the source does not meet all inclusion criteria. We took a conservative approach to exclusion: sources for which a decision could not be made in the first phase were retained for evaluation in the second phase.

All steps were performed by two authors independently, and disagreements solved via discussion with a third author. We assessed the authors' agreement with Cohen's kappa coefficient (Emam 1999). The kappa coefficient considers the observed frequency of agreement relative to the expected probability of agreement, assuming authors decide randomly and independently.

Finally, we performed forward and backward snowballing (Wohlin 2014) on the academic sources, i.e., we reviewed in the same way as described above all references and citations in the papers' introduction, related work, and discussion sections (or equivalent sections with different titles) if these existed. For the gray literature sources, we instead checked whether contained links to other resources refer to tools that are in scope.

We examined the identified sources in detail to identify all presented and mentioned tools for microservice architecture recovery. Specifically, we looked for any references to source code repositories, web applications, Docker images, or other ways of providing a tool. In line with the objective of the study, only tools that follow a static analysis approach or hybrid approach where the static part can be run independently were considered. The first author performed the identification. In cases where no tools were found in a source, the second author checked the source as well for confirmation. The resulting list of static analysis microservice architecture recovery tools serves as the answer to **RQ1**.

2.2 Tool characterization

For all identified tools, we examined the available information (corresponding publication, code repository, tool description, etc.) to identify their general properties (platform, language, static/hybrid approach, output format, etc.), as was done by Bakhtin et al. (2024) (step ② in Fig. 1). Here, those characteristics extracted by the tools that go beyond the basic architecture of the analyzed systems were of particular interest. The results are presented in a matrix listing all identified tools as well as the characteristics they extract. This matrix later determined which tools are compared with each other based on each characteristic. In addition to guiding the comparison of tools' extraction accuracy, the created matrix is also the basis to answer **RQ2** and **RQ3**.

2.3 Extension of dataset

To compare the identified tools' correctness in architecture recovery under controlled circumstances, they needed to be executed on the same dataset. We used a dataset of 17 data-flow diagrams (DFDs) of open-source microservice applications (Schneider et al. 2023) for this purpose. To the best of our knowledge of the research field, it is the only dataset suited for this purpose. We are not aware of any other collection of manually created and verified architectural models in this form. Table 1 lists the applications in the dataset along with their number of components, connections, and endpoints. They are typical, small- to medium-sized open-source microservice applications written in Java with a focus on the Spring framework. It has been reported, that Java is the most popular language for developing microservice applications (JetBrains 2022) and that Spring is the most used frame-

Table 1 Microservice applications in the used dataset, their numbers of components (**Cp**), connections (**Cn**), and endpoints (**E**)

App.	GitHub Repository	Cp	Cn	E
1	anilallewar/microservices-basics-spring-boot	12	29	8
2	apssouza22/java-microservice	15	34	17
3	callistaenterprise/blog-microservices	17	42	10
4	ewolff/microservice	7	13	15
5	ewolff/microservice-kafka	8	12	11
6	fernandoabcampos/spring-netflix-oss-microservices/	11	24	17
7	georgwittberger/apache-spring-boot-microservice-example	5	6	6
8	jferrater/tap-and-eat-microservices	9	16	11
9	koushikkothagal/spring-boot-microservices-workshop	5	6	7
10	mdeket/spring-cloud-movie-recommendation	11	18	17
11	mudigal-technologies/microservices-sample	15	34	3
12	piomin/sample-spring-oauth2-microservices	8	13	2
13	rohitghatol/spring-boot-microservices	11	26	8
14	shabbirdwd53/springboot-microservice	9	18	6
15	spring-petclinic/spring-petclinic-microservices	12	28	10
16	sqshq/piggymetrics	17	37	10
17	yidongnan/spring-cloud-netflix-example	10	29	2

work for Java microservice applications (JRebel 2022). The applications in the dataset were selected from sources in the literature as well as popular repositories on GitHub. According to the dataset's publication, established design patterns for microservice applications using the Java Spring framework are prevalent in the dataset. We preserved all the repositories from the dataset in our own forks[12].

The DFDs in the dataset depict the applications' architecture as well as additional properties. Nodes in the DFDs represent components of the applications, i.e., (internal and infrastructural) microservices, databases, and external entities; edges represent connections between any two components. As such, the nodes and edges were used as ground truth for the basic architecture (i.e., RQ4.1 and RQ4.2 are answered based on the tools' accuracy in extracting these characteristics).

To answer RQ4.3, an extension of the dataset was needed. As will be presented later in Section 3.4, the characteristic *endpoints* was the only additional one to components and connections that could be part of the comparison. In this context, endpoints describe explicitly specified endpoints of components to which RESTful HTTP requests can be made. The DFDs in the dataset contain extensive annotations that represent security mechanisms, deployment information, and other system properties, including information about endpoints. However, the information about endpoints was not complete. Therefore, the DFDs needed to be extended concerning this characteristic to serve as ground truth for evaluating the tools' extraction correctness. A look into the tools that have endpoints in their extraction scopes and an investigation into possibilities for implementing endpoints revealed that a number of Java annotations indicate such endpoints. Specifically, identifying the annotation `@RequestMapping` and its related, more specific annotations for a single HTTP method (`@PutMapping`, `@GetMapping`, and so on) and the annotation `@Reposito-`

[12] https://github.com/M3SOulu/EMSE2025SAR-Benchmarks

`ryRestResource` in the code is sufficient to manually create the ground truth required for this work.

Due to this simple and unambiguous nature of the identification of endpoints, a single author searched for relevant annotations in the source code of all applications in the dataset and extended the DFDs accordingly. A second author checked the newly added endpoints for correctness and found no errors.

We identified 63 additional endpoints following this process and added them to the DFDs in the dataset. As a result, the dataset now contains exhaustive information about endpoints specified with the considered Java annotations.

Since some of this paper's authors are also authors of the initial dataset, we have integrated the described changes directly. All means of obtaining the dataset have been updated to the new data (the corresponding website[13] and GitHub repository[14] contain the new data, and we added a new version containing the new data to the persistent repository at Zenodo[15]).

2.4 Execution of identified tools

To obtain outputs from the identified tools for their evaluation, they were run on the applications in the dataset. Naturally, the tools needed to be executed successfully to create outputs. There were some obstacles in terms of reproducibility, i.e., it was not trivial to execute some tools. To achieve a fair comparison, a methodology for executing them was established (step ④ in Fig. 1) that ensures that the effort invested into attempting to run each tool is comparable. We first attempted to run a tool based on available instructions (documentation, information in the source, etc.), possibly on a virtual machine if specific requirements for the execution environment were mentioned. Where this was not successful, we analyzed the code for indicators of how to run the tool (code comments, hints by identifiers, error messages during execution, etc.). If all steps failed, we abandoned the tool and excluded it from the comparison. As an indicator of a successful execution, we checked whether the tool produced any output in the form of extracted characteristics or status information signaling a completed run.

2.5 Comparing tools' accuracy

The metrics precision, recall, F1-score, and execution time were used as quantitative measures for comparing the tools. These are common and objective metrics used for such evaluations. Although we do not dictate a specific use case for the tools, their ability to perform their core functionality correctly is the most important basis for evaluation. Ideally, an architecture recovery tool should extract all existing characteristics in its extraction scope and not falsely produce results for more than these. Precision and recall serve as measures to indicate these two aspects, the F1-score serves as harmonic mean between the two to allow a clear comparison based on a single score. The relevance of the execution times is more dependent on the intended use case, but is important for most scenarios as well. Lightweight

[13] https://tuhh-softsec.github.io/microSecEnD/

[14] https://github.com/tuhh-softsec/microSecEnD

[15] https://zenodo.org/records/7714926

static analysis tools that show quick execution times lend themselves to being integrated into automated pipelines such as fast-paced CI/CD pipelines. For example, the output of architecture recovery tools could be used by model-based analysis tools in a deployment pipeline.

To quantify the tools' output, the number of correctly extracted characteristics (true positives, TP), the number of falsely extracted characteristics (false positives, FP), and the number of undetected characteristics (false negatives, FN) were manually counted by comparing the output to the ground truth (step ⑤ in Fig. 1). Since the tools have different output formats, quantifying the results manually was deemed the safest method for a correct representation.

The process was performed by two authors independently, and disagreements were solved in discussion with a third author. Precision, recall, and F1-score were calculated from these measures with the following formulas:

$$\text{Precision} = \frac{\text{Correct characteristics}}{\text{Correct characteristics} + \text{False characteristics}}$$

$$\text{Recall} = \frac{\text{Correct characteristics}}{\text{Correct characteristics} + \text{Undetected characteristics}}$$

$$\text{F1-score} = \frac{2 * \text{Correct characteristics}}{2 * \text{Correct characteristics} + \text{False characteristics} + \text{Undetected characteristics}}$$

RQ4 is answered based on the above measures. Specifically, we compared different subsets of the complete list of tools against each other. The overview of each tool's extracted characteristics (see Section 2.2) determined which tools were compared with each other. For each characteristic, we compared all tools that are supposed to extract it in extracting this characteristic concerning their accuracy.

Additionally to evaluating the tools via these measures, we also investigated the reasons for tools' detection errors. For this, two authors manually examined false positives and false negatives that occurred in order to understand their causes by tracing the analysis to the source code.

The tools' execution times were measured by first creating Bash scripts that execute a tool on all applications on which it can be run. The scripts consider only the analysis itself, i.e., they exclude the possibly needed cloning of repositories or compilation of the analyzed applications. The scripts were then run, and the execution times measured using the `time` terminal command. Each tool's script was executed ten times and the average execution time calculated.

2.6 Evaluating combinations of tools

Additionally to evaluating the individual tools, we investigated whether it is possible to improve their results by combining multiple tools. **RQ5** is answered based on this evaluation and its comparison to the results of the individual tools. Tools could show synergies concerning the scope of extracted characteristics and concerning the applied detection techniques. It has been reported, that many developers already use multiple complementary analysis tools in their software development process. For example, in a large-scale study with practitioners by Do et al. (2022), 36.8% of participants reported using only a single

analysis tool, the other 63.2% consequently more than one, and in a study with practitioners by Vassallo et al. (2020), 31 out of 56 participants (55%) reported using multiple static analysis tools. It is therefore reasonable to assume, that combinations of tools could be used by practitioners. Further, we suggest that developers of architecture reconstruction tools could create tools that combine multiple analysis approaches in a single implementation. Therefore, we evaluated the performance of tool combinations by combining the individual tools' results. The combinations were selected based on the individual results such that the used metrics (precision, recall, and F1-score) are maximized.

In this context, the tools' results are combined with logical "AND" and "OR" operators on the individual characteristic level, indicated in the following as subscript annotations to the tool combinations' IDs. For each individual characteristic, combining the results of two or more tools with a logical "AND" means that the characteristic is successfully extracted (TP) if all tools have done so. Otherwise, it counts as FN. A FP is counted if all tools falsely detected the same individual characteristic. Note that there can also be the case that only one of the tools in a combination successfully produced results for an application, in this case, its results count directly for the combination. For the combinations with a logical "OR", it is sufficient if one of the combinations' tools had detected an individual characteristic to count as TP for the combination, and only if none of the tools detected it does it count as FN. Here, all FPs of the individual tools are considered and those falsely detected by all tools only count as one FP. Table 2 summarizes the above description. Precision, recall, and F1-measure are calculated in the same way as for the individual tools.

3 Tools identified via a multivocal literature review

The multivocal literature review was executed following the methodology described in Section 2.1 (step ① in Fig. 1). The process and identified tools are presented in the context of formally and informally published sources, with a description of the tools and their characteristics.

3.1 Review of informally published (Gray Literature) sources

Figure 2 (right side) visualizes the process of the conducted gray literature review. Searching for the formulated search string yielded 315 initial results across the four considered

Table 2 Definitions of true positives (TP), false negatives (FN), and false positives (FP) for "OR" and "AND" combinations

Combination	Measure	Description
OR	TP	*Any* tool *correctly* detected the characteristic.
	FN	*No* tool *correctly* detected the characteristic.
	FP	*Any* tool *falsely* detected the characteristic.
AND	TP	*All* tools *correctly* detected the characteristic.
	FN	*Any* tool did *not correctly* detect the characteristic.
	FP	All tools *falsely* detected the characteristic.

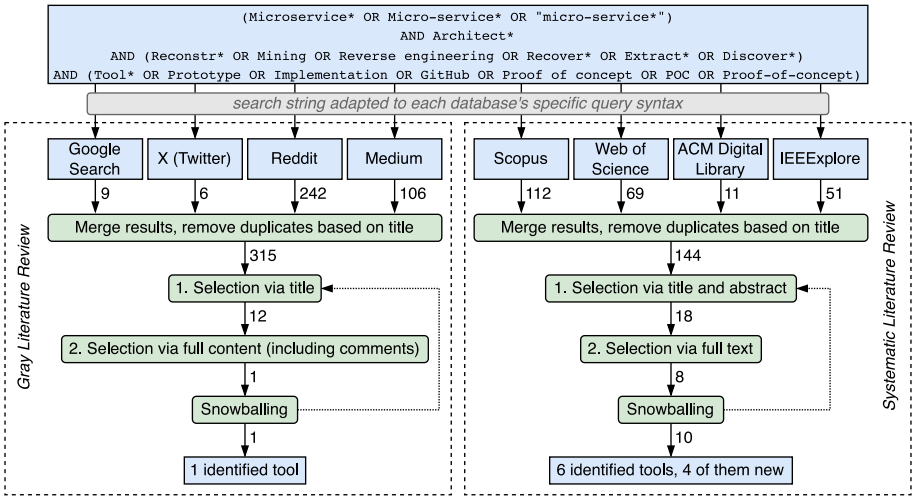

Fig. 2 Results of the systematic literature review of the formally published literature (left) and the gray literature review of the informally published literature (right)

sources. Most results were found on Reddit (204) and on Medium (106). The results from Reddit contained mostly posts asking for advice, e.g., with architecting microservice applications, implementing specific technologies, visualizing an architecture, or a reference to a tool that could perform architecture reconstruction. The search on Medium resulted in many unrelated posts without any connection to microservices or even computer science. Those results that were more relevant contained mostly guides on specific implementations or technologies.

As shown in Fig. 2, twelve results were retained after applying the exclusion criteria in the first phase, i.e., after reading the results' titles and descriptions. After reading the complete results in the second phase, all but one result were excluded. The only result that is in the scope of the study contains a reference to a proprietary tool that meets the inclusion criteria. The tool is not open-source, but a free use plan exists. While we would not be able to perform any debugging steps in case of execution problems (see step ④ in Fig. 1), the tool meets the defined scope of our study, and we thus opted to include it in the further process.

The calculated Cohen's Kappa for interrater agreement showed almost perfect agreement for the first phase ($\kappa = 0.84$) and perfect agreement for the second phase ($\kappa = 1.0$) according to Landis and Koch's classification (Landis and Koch 1977). The reason for such a high agreement was due to the rather unambiguous inclusion decision to be made – identifying a reference to a tool that is in the scope of our study or concluding that such a reference is not given is a straightforward task when assessing the complete source.

3.2 Review of formally published (Academic) literature

For the identification of tools in the academic literature, we performed a systematic literature review following the methodology described in Section 2.1. Figure 2 (left side) shows the intermediate results of each step. The reported numbers correspond to results that were published after the search date reported by Bakhtin et al. (2024), February 23rd, 2023. The

search in the four formal databases yielded 144 results. After applying the inclusion/exclusion criteria based on the articles' title and abstract in the first phase, 18 articles remained. Applying the criteria on the articles' full content in the second phase retained 8 articles. We added one source known to us, which we know to contain a tool in the scope of our study (*MicroGraal* Hutcheson et al. 2024). It was not found via the search because it was only recently presented. A following forwards and backwards snowballing revealed one further candidate.

From the ten sources, six tools could be identified, however, two of those had already been found with the previous study. Their occurrence in both literature reviews that should have mutually exclusive results due to the considered time periods is explained by pre-prints that had been published before the cut-off date and the final publications being published after. Thus, four new tools have been identified in the SLR, compared to the previous list of tools.

The Cohen's Kappa coefficient for interrater agreement indicated substantial agreement for both phases ($\kappa = 0.77$ for both) (Landis and Koch 1977).

3.3 Identified tools

The analysis of the selected articles from the formally published literature and resources from the gray literature resulted in the identification of five tools that are in the scope of our study and were not identified by Bakhtin et al. previously (Bakhtin et al. 2024). Together with the tools identified by them that fit the selection criteria of our study, 13 tools are in the scope of our study. Table 3 presents the final list of twelve open-source and one proprietary tools. We preserve forks of all OSS tools[16].

> **Answer to RQ1:** Via the multivocal literature review, we identi ed 13 static analysis tools to recover the software architecture of microservice applications. Twelve of the tools are open source, one is proprietary. Nine tools are further evaluated in the comparison of tools.

The tools' source code and corresponding publications were analyzed in detail to identify characteristics they are expected to detect (step ② in Fig. 1). We also attempted to run each of the identified tools to obtain results of the architecture recovery (step ④ in Fig. 1). In the following, each tool is introduced and the process of executing them is described. Where tools were excluded from the further comparison study, the reasons are given in detail.

Authz-flow-analysis Abdelfattah et al. (2023) proposed a human-centric but tool-supported method to analyze the access rights of microservice applications. Specifically, the tool *authz-flow-analysis* detects all endpoints in a microservice application and determines, which of the operations create, read, update, or delete (CRUD) is associated with each endpoint. According to the authors, the authorization security policies in a microservice system are based on CRUD operations, and their automatic detection therefore eases the analysis process for a human. The tool analyzes an application's source code and detects endpoints

[16] https://github.com/M3SOulu/EMSE2025SAR-Tools

Table 3 Identified static analysis architecture recovery tools

Tool	GitHub Repository	Pub.	Comp.
authz-flow-analysis	cloudhubs/ authz-flow-analysis	Abdelfattah et al. (2023)	□
Attack Graph Generator	tum-i4/ attack-graph-generator	Ibrahim et al. (2019)	■
Code2DFD	tuhh-softsec/code2DFD	Schneider and Scandariato (2023)	■
MicroDepGraph	clowee/MicroDepGraph	Rahman et al. (2019)	■
MicroGraal	cloudhubs/ graal-prophet-utils	Hutcheson et al. (2024)	■
microMiner	di-unipi-socc/ microMiner	Muntoni et al. (2021)	■
microTOM	di-unipi-socc/ microTOM	Soldani et al. (2023)	□
Prophet	cloudhubs/prophet	Bushong et al. (2021)	■
Prophet2	cloudhubs/prophet2	Schiewe et al. (2022)	□
protoc-gen-scip	CUHK-SE-Group/ protoc-gen-scip	Fang et al. (2023)	□
RAD	cloudhubs/rad	Das et al. (2021)	■
RAD-source	cloudhubs/rad-source	Das et al. (2021)	■
Tool	**Website**		
Contextmap	https://www.context-map.io/		■

Pub. = Publication; **Comp.** = considered in comparison of tools, reasons for exclusion are discussed in the presentation of tools below

and the CRUD operation that can be performed via them. From this information, the application's call graph is constructed.

The detection of CRUD operations in *authz-flow-analysis* is built on top of the identification of endpoints via a library called `source-code-parser` that creates a language-agnostic abstract syntax tree of the analyzed application and that is also used in another tool by the same authors, *Prophet2*. In the examination of the other tool, we succeeded in running the functionality of this library but not the rest of the analysis.

This tool is the only one in the list of identified tools that has the detection of CRUD operations in its extraction scope. We could not have compared its performance to others and therefore did not proceed to create the ground truth needed for this characteristic.

Attack Graph Generator Ibrahim et al. (2019) presented their tool *Attack Graph Generator*, which automatically generates attack graphs for microservice applications. Architecture extraction is performed as preliminary for the generation of the attack graphs, i.e., only a means to perform other analysis. The tool's approach is based on methods from the field of computer networks.

For this study, only one of the tool's three main components, the "Topology Parser", is relevant. It parses Docker Compose files to extract the analyzed applications' architecture, which is then used in the further analysis that is not relevant for this study. The Topology Parser extracts components and connections of analyzed applications. The tool should also detect privileges granted to some Docker containers, however, we did not observe any indications of this in the results.

The tool's repository contains a shell script that installs all required dependencies and then executes a Python script that runs the main functionality. The documentation states that the tool was only tested on a virtual machine running Ubuntu 16.04 LTS. Unfortunately, this operating system has reached its end of service since 2021 and a virtual machine that we set up with it could not execute programs vital for running the *Attack Graph Generator*. We instead attempted to run the tool on a MacOS based machine, which resulted in errors. However, a minor debugging effort revealed that the crashes had occurred after the Topology Parser component had finished its execution. The creation of the topology graph of the analyzed application had finished successfully. We accepted the failing of the part of the analysis that is not relevant for our study and proceeded with the created topology graphs. Since the setup described in the tool's documentation could not be replicated, the results might have been affected, but no indications of this were observed and the generated topology graphs show meaningful results.

Code2DFD presented by Schneider and Scandariato (2023) is an approach and tool for the extraction of security-rich dataflow diagrams (DFDs) from the source code of microservice applications. The extracted DFDs depict the architecture consisting of nodes and edges, as well as additional annotations that represent system properties such as implemented security mechanisms, deployment information, and other. The approach is based on the detection of keywords in the analyzed applications' source code. Detected keywords serve as evidence for proving the existence of the characteristics they implement. Based on this technique, the approach can also create traceability information for the model items, i.e., the place of the evidence for the item's existence in the source code. For the work presented in this paper, the characteristics components, connections, and endpoints are relevant and can be compared to other tools.

Code2DFD is implemented in Python and available on GitHub. After downloading the source code, the tool is executed via the terminal by providing the link to the analyzed application's GitHub repository. Alternatively, applications can be analyzed locally if the source code is available there, and the tool can also be run as a Flask server and the architecture extraction triggered via API requests. *Code2DFD*'s repository contains instructions on the usage of the different possibilities. We encountered no obstacles in the process of running the tool and it can analyze all applications in the used dataset.

ContextMap is the only proprietary tool that we identified. It is marketed as an "automated documentation solution" for software development, intended to solve issues stemming from manual documentation. The documentation is centered around architecture recovery. The tool is made to be integrated into CI/CD pipelines and in this case scans the application at every commit. A dashboard visualizes the recovered architecture and provides information about the development process such as releases, deployments, development team structure, and similar. The tool only works for applications using Maven as a build system.

Aside from the commercial version, *Contextmap* also offers a free "Community" version, which has some limits on the size of analyzable applications, number of users, etc., but offers the full functionality. It can be obtained and installed as a Docker image from Docker Hub without any problems. To analyze an application, the `pom.xml` file of each individual microservice needs to be modified by adding *ContextMap* as a plugin. The architecture extraction is initiated via the terminal for every microservice individually (in the case that the tool is not integrated into a CI/CD pipeline) and the results are shown in a dashboard when running the Docker image locally.

We contacted the *ContextMap* team because the tool's documentation states, that the tool should only require changing the root Maven file for multi-module projects and that the analysis does not need to be performed for each microservice individually. Further, we did not see any connections being detected. We received an answer giving us the above-described solution, and also the information that the tool only detects connections implemented via Spring's *FeignClient* or *HttpExchange*. The detection of other connections can be enabled by manually annotating clients in the source code with a custom annotation. But most connections in the applications in our dataset use *FeignClient* or *HttpExchange*, and we deemed a manual annotation to not be in line with the executed study.

MicroDepGraph is presented by Rahman et al. (2019) in the context of creating a dataset of microservice applications. It visualizes the dependencies between the microservices of microservice applications to show a graph of components and connections. The tool analyzes Docker Compose files to recover the required information, and can detect API calls between services directly in Java source code.

MicroDepGraph is provided as a Java project on GitHub. After building it with Maven, it can be executed on applications locally, i.e., the source code of applications to be analyzed need to be cloned from GitHub or obtained otherwise.

When we executed the tool, it crashed when trying to access a Neo4J database, which we had not set up because the documentation does not ask for it. However, the tool saves the recovered architecture (called "topology graph" there) of the analyzed application as an .svg file before attempting to access the database. According to the tool's corresponding paper, the information contained in the .svg file and database are the same. Therefore, we did not proceed with trying to solve this issue but see the creation of the .svg file as successful execution of the tool and based our analysis on it.

MicroGraal Hutcheson et al. (2024) proposed a framework to perform static reconstruction of microservice systems based on compiled bytecode using the GraalVM native image. The use of bytecode instead of source code or deployment information is what makes

this approach distinct from most others. The authors note, that applications' source code might not be available in certain scenarios due to security or proprietary rights reasons. *MicroGraal* focuses on Java applications using the Spring framework.

Applications to be analyzed need to be compiled with the authors' custom version of the GraalVM native image, which is also available on GitHub. *MicroGraal* is then executed with the output of the compilation and the list of microservice base package names as input and can detect instances of Java annotations, leading to the recovery of endpoints and REST calls of the microservices. A "communication graph" and a "system context map" are created from the detected information.

We were successful in executing the tool on the application *Train Ticket*[17], thereby replicating the validation reported by the authors in their publication. However, the tool produced empty results for all applications in the dataset we used in our study. All applied debugging efforts could not resolve this issue.

microMiner presented by Muntoni et al. (2021) is a hybrid approach to recover microservice applications' architecture via Kubernetes deployment files. The static part of the analysis is the first step in the approach and can be executed independently, therefore, we included it in our study. The static mining creates a "partial topology graph", which contains the application's microservices. Connections between them are only extracted in later analysis steps dynamically and are therefore not considered in our comparison.

The tool is implemented in Python. After cloning the tool's repository, it can be run on local applications' source code directly. For applications on GitHub, their source code needs to be cloned prior to the analysis. We faced some difficulties with suppressing the tool's dynamic analysis part, but received support from the authors upon contacting them. Still, some minor issues occurred in the parsing of Kubernetes files, based on some text comprehension commands that did not consider all legal structures of Kubernetes files. We adjusted the tool slightly and could then run it successfully. Also, we leveraged results from previous work (Cao et al. 2024) in which some applications in the dataset were extended to be deployed via Kubernetes. The Kubernetes manifests created for this were added to the dataset to make more applications analyzable with *microMiner*.

The output of *microMiner* is a YAML file following the *microTOSCA* format[18], an extension of the TOSCA modelling standard specifically for microservice applications. The *microTOSCA* files can be visualized with another tool, *microFreshener*[19]. This visualization, however, was broken when we ran *microFreshener*. Instead of investigating this issue, we used the YAML files directly to perform our analysis – a less convenient process but possible to do manually since the format lists components in a structured way.

microTOM presented by Soldani et al. (2023) is a hybrid tool that recovers microservice applications' architecture from their Kubernetes deployment files. It was developed by the same authors as *microMiner* (see paragraph above) and is closely related to it, following the

[17] https://github.com/FudanSELab/train-ticket/

[18] https://github.com/di-unipi-socc/microTOSCA

[19] https://github.com/di-unipi-socc/microFreshener

same detection technique and adding more elaborate information to the created graph. The static analysis part of the approach, i.e., the parsing of Kubernetes deployment files, follows the same process as in *microMiner*. One of the authors confirmed, that the static part of the analysis is equivalent between both tools and that the same results are achieved. Consequently, we did not consider this tool in the following comparison, since the evaluated part of it would have been a duplicate. The interested reader may refer to both tools to compare the dynamic parts of the analysis.

Prophet by Bushong et al. (2021) analyzes the applications' source code to recover endpoints and connections between them. Apart from detecting Java annotations to this end, it also relies on "enterprise standards", e.g., by detecting function names that are commonly chosen as identifiers by developers. The tool also has the applications' components in its extraction scope. It detects them via the folder structure of the analyzed code (which can be local or on GitHub). Specifically, all folders in the root of the provided directory are detected as components of the application. An additional extracted characteristic is a "context map", which is created by detecting all classes in the application and identifying those that act as entities, but no further information is provided as to what such a context map looks like or represents. Overall, the specific workings of the tool are not clear by either the publication or the code repository.

Prophet is built on top of *RAD* / *RAD-source* (see below) by the same team that created these tools. It is available as a library and as a microservice wrapper around that library, which accepts a POST request with a path to the analyzed project as a parameter. It analyzes the code locally, i.e., if the provided path is a GitHub repository, this will be cloned first.

Prophet2 is a migration of the *Prophet* tool (see above) to the Rust programming language presented by Schiewe et al. (2022). It further enhances the analysis technique from being Java-specific to language-agnostic by first representing the analyzed application as a language-agnostic abstract syntax tree (LAAST). In these, different characteristics can theoretically be identified, which is demonstrated in the paper with a case-study on two applications. The tool implemented for this can detect components, connections, and endpoints, where the detection is largely based on leveraging naming conventions of identifiers in the code.

The tool is centered around a separate submodule called "source-code-parser", which can also be run as a stand-alone service, and is responsible for creating the LAASTs. We could successfully compile this submodule after some difficulties with outdated versions, and could then generate LAASTs by sending requests containing a list of the files to be analyzed to it. However, *Prophet2* itself did not compile and could not be fixed by us, despite considerable debugging effort. The tool's repository contains no documentation or even information on how to run it, all above information stems from our analysis of the source code. We reached out to the authors of the tool but could not receive any support, because the tool has not been used in a long time and the main developers of the tool have moved on from academia. We decided that abandoning the tool from our study was reasonable, based on the invested effort and inability to receive further support.

protoc-gen-scip is a plugin for the `protoc` compiler developed by Fang et al. (2023). It targets microservice applications using the RPC protocol for inter-service communication instead of REST API calls.

The approach uses SCIP[20] to represent code dependencies of individual microservices via language-specific tools (e.g., `scip-go`, `scip-java`) that need to be installed. These results are used as input for *protoc-gen-scip*, which first creates a single SCIP index for all microservices by matching RPC calls between services and then creates a system overview as a graph out of it.

All applications in the dataset used in this paper use REST calls for communication between services. The authors of *protoc-gen-scip* themselves note, that there is a lack of available applications using the RPC protocol. For an evaluation in their paper, they use one open-source application and one application they created for this purpose. We decided to abandon this tool from the further study, because the creation of a ground truth for a number of new applications would take a large effort and not yield a fair comparison of tools, since there would be no overlap between the applications analyzed by *protoc-gen-scip* and those analyzed by any other tool.

RAD is a tool proposed by Das et al. (2021) that is part of a larger approach to construct a role-based access control model (i.e., a mapping of the access roles required to interact with endpoints)and determine inconsistencies in it. The architecture recovery is the initial step in the process. The tool detects endpoints implemented with certain Java annotations and identifies connections between them. The resulting architecture is then further used for the access control analysis in a human-in-the-loop approach.

The tool is provided as a Java project on GitHub and can be built with Maven. Then, it can be run as a microservice and the analysis be executed by sending an API request to it. It takes the bytecode of the applications as input, therefore, any application to be analyzed needs to be compiled first.

The tool successfully identified endpoints and connections for the application that was used as an evaluation in the corresponding paper. For the applications in the dataset used in this paper, however, only endpoints were detected.

RAD-source The authors of *RAD* (see paragraph above) provide *RAD-source* as a second tool in the same publication (Das et al. 2021). It is an alternative to *RAD* that takes the analyzed applications' source code as input instead of the compiled bytecode. The recovered architecture should be the same as that of *RAD*, and the integration of *RAD-source* in the larger approach of access control analysis is the same as well. The tool is also provided as a Java project that is built and then run as a microservice that takes API requests to trigger the analysis. In contrast to *RAD*, *RAD-source* identified some connections in the analyzed applications.

[20] https://github.com/sourcegraph/scip

The description of the identified tools and the characteristics they extract serve as the answer to RQ2. In summary:

Answer to RQ2: There are five characteristics additionally to the applications' components and connections that are extracted by one or more of the 13 identified tools: *endpoints*, *CRUD operations*, *attack graphs*, *security and other properties*, and *context maps*.

3.4 Extracted characteristics

The above description of the identified tools and the specific architecture reconstruction they perform yields a tool characterization matrix (step ② in Fig. 1), indicating for each tool which characteristics it is intended to extract from the applications it is used to analyze. In the remainder of this article, the group of characteristics a tool should extract is called its *extraction scope*. Table 4 presents the tool characterization matrix, showing each tool's extraction scope.

Table 4 shows, that the most common characteristic in the tools' extraction scopes are connections between components. Eleven tools are intended to extract it, only two are not. Interestingly, only seven of the tools are meant to extract those components in the first place. The others require the list of components to be provided as input or the tool to be executed for each component individually, which we do not consider to be an extraction of the components. Concerning the five characteristics that are extracted beyond the basic architecture, seven tools consider the microservices' REST endpoints and two related tools provide a Context map of all system entities. The other three characteristics (CRUD operations, attack graphs, and security and other system properties) are in the extraction scope of a single tool each.

Table 4 Tool characterization matrix

	Characteristic						
Tool	Components	Connections	Endpoints	CRUD operations	Attack graphs	Security and other properties	Context map
authz-flow-analysis	□	■	■	■	□	□	□
Attack Graph Generator	■	■	□	□	■	□	□
Code2DFD	■	■	■	□	□	■	□
MicroDepGraph	■	■	□	□	□	□	□
MicroGraal	□	■	■	□	□	□	■
microMiner	■	□	□	□	□	□	□
microTOM	■	□	□	□	□	□	□
Prophet	■	■	■	□	□	□	■
Prophet2	■	■	■	□	□	□	□
protoc-gen-scip	□	■	□	□	□	□	□
RAD	□	■	■	□	□	□	□
RAD-source	□	■	■	□	□	□	□
Contextmap	□	■	■	□	□	□	□

Table 4 provides the answer to RQ3:

Answer to RQ3: The most commonly considered characteristic by the tools are connections between components (eleven out of 13 tools extract them). The systems' components (seven tools) and REST endpoints (eight tools) are also prevalent in the extraction scopes of the tools. Other characteristics are considered by not more than two tools for the same characteristic.

4 Comparison of tools

Comparing the tools' accuracy in performing their specific architecture extraction is a novelty of this work that fills a gap in the related literature published so far. As discussed in Section 3.3, four tools are excluded from the comparison. Table 5 lists the excluded tools and the reasons for the exclusion. The two tools *authz-flow-analysis* and *microTOM* are not considered because they realize the architectural reconstruction with other identified tools and do not qualify as separate tools for the sake of this study. The tools *Prophet2* and *protoc-gen-scip* are also excluded, as described.

After executing the nine tools that are considered in the comparison on all applications in the dataset, we manually quantified the results following the methodology presented in Section 2. The following sections present the results of this process per extracted characteristic and compare those tools that have that characteristic in their extraction scope. We note that judging the results depends highly on the usage scenario and that the reader should rely on the raw metrics rather than our description of what is considered a high or low result when considering adopting one of the tools for a specific scenario. Nevertheless, we use such classifying terms to better show the comparison of the tools' observed correctness without further context.

4.1 Individual tools

Herein, we present the comparative results of the identified tools' extraction accuracy for different characteristics.

Components Five of the identified tools extract the analyzed systems' components, i.e., their individual microservices. Table 6 shows the observed results. Three tools (*MicroDepGraph*, *Code2DFD*, and *MicroMiner*) show a high precision with results of 1.0, 0.97, and 0.97 in this metric, respectively. *Attack Graph Generator* and *Prophet* show higher amounts of false positives, resulting in a precision of 0.75 and 0.56, respectively. *Code2DFD* is able to extract almost all components, with only five false negatives out of 182 components in the ground truth, resulting in a recall of 0.98. *Attack Graph Generator* and *MicroDepgraph*

Table 5 Tools excluded from the comparison and the reasons for the exclusion

Excluded Tool	Reason
authz-flow-analysis	No overlap in extraction scope to other tools.
microTOM	Duplicate extraction technique of microMiner
Prophet2	Could not run the tool.
protoc-gen-scip	No overlap in extraction scope to other tools.

Table 6 Tools' extraction results for the characteristic (a)*components*, (b) *connections*, and (c) *endpoints*

(a) Components

Tool	**GT**	**TP**	**FP**	**FN**	**P**	**R**	**F1**
Attack Graph Generator	144	123	41	21	0.75	0.85	0.8
Code2DFD	182	178	5	4	0.97	**0.98**	**0.98**
MicroDepGraph	110	84	0	26	**1.0**	0.76	0.87
MicroGraal	—	—	—	—	—	—	—
microMiner	57	32	1	25	0.97	0.56	0.71
Prophet	182	27	21	155	0.56	0.15	0.23
RAD	—	—	—	—	—	—	—
RAD-source	—	—	—	—	—	—	—
Contextmap	—	—	—	—	—	—	—

(b) Connections

Tool	**GT**	**TP**	**FP**	**FN**	**P**	**R**	**F1**
Attack Graph Generator	324	279	432	45	0.39	**0.86**	0.54
Code2DFD	385	320	27	65	0.92	0.83	**0.87**
MicroDepGraph	245	124	2	121	**0.98**	0.51	0.67
MicroGraal	173	0	0	173	n/a	0	0
microMiner	—	—	—	—	—	—	—
Prophet	385	4	2	381	0.67	0.01	0.02
RAD	259	0	0	259	n/a	0	0
RAD-source	385	7	2	378	0.78	0.02	0.04
Contextmap	206	0	0	206	n/a	0	0

(c) Endpoints

Tool	**GT**	**TP**	**FP**	**FN**	**P**	**R**	**F1**
Attack Graph Generator	—	—	—	—	—	—	—
Code2DFD	160	86	13	74	0.87	0.54	0.66
MicroDepGraph	—	—	—	—	—	—	—
MicroGraal	121	0	0	121	n/a	0	0
microMiner	—	—	—	—	—	—	—
Prophet	160	0	0	160	n/a	0	0
RAD	132	90	6	42	**0.94**	**0.68**	**0.79**
RAD-source	160	86	11	74	0.89	0.54	0.67
Contextmap	91	0	0	91	n/a	0	0

(d) All characteristics in tool's extraction scope;
Filled box under **Cp**/**Cn**/**E** = components / connections / endpoints are in tool's extraction scope.

Tool	**Cp**	**Cn**	**E**	**GT**	**TP**	**FP**	**FN**	**P**	**R**	**F1**
Attack Graph Generator	■	■	□	468	402	473	66	0.46	**0.86**	0.60
Code2DFD	■	■	■	727	584	45	143	0.93	0.80	**0.86**
MicroDepGraph	■	■	□	355	208	2	147	**0.99**	0.59	0.74
MicroGraal	□	■	■	295	0	0	295	n/a	0	0
microMiner	■	□	□	57	32	1	25	0.97	0.56	0.71
Prophet	■	■	■	727	31	23	696	0.57	0.04	0.08
RAD	□	■	■	391	90	6	301	0.94	0.23	0.37

Table 6 (continued)

RAD-source	□	■	■	545	93	13	452	0.88	0.17	0.29
Contextmap	□	■	■	297	0	0	297	n/a	0	0

GT = number of individual characteristics in the applications that the tool should be able to analyze; **TP**/**FP**/**FN** = observed true positives / false positives / false negatives; **P**/**R**/**F1** = calculated precision / recall / F1-score; Dashes (—) = characteristic is not in tool's extraction scope

with a recall of 0.85 and 0.76, respectively, also perform well. *MicroMiner* shows a recall of 0.56 and *Prophet* only identifies few components (recall of 0.15). Looking at the F1-score, *Code2DFD* achieves the best result with a value of 0.98. The next-best tools are *MicroDepGraph* with a score of 0.87 and *Attack Graph Generator* with a score of 0.80, followed by *microMiner* with 0.71. *Prophet* achieves a low score of 0.23.

In summary, we observed the highest recall and one of the highest precision values for *Code2DFD* for this characteristic, resulting in the best F1-score of 0.98. *MicroDepGraph* showed no false positives, which resulted in a perfect precision, but a lower recall. *Attack Graph Generator* shows good results in all metrics, while *microMiner* also has a high precision but low recall. *Prophet* shows the worst accuracy out of the tools concerning this characteristic.

Answer to RQ4.1: Table 6a presents the tools' observed accuracy in extracting the analyzed applications' components. *Code2DFD* showed the highest F1-score of 0.98, *MicroDepGraph* a score of 0.87, *Attack Graph Generator* a score of 0.80, and *microMiner* a score of 0.71.

Connections Eight of the identified tools have the analyzed applications' connections in their extraction scopes. Table 6b presents the observed results. It shows that only three tools were able to do so somewhat effectively. Three tools (*MicroGraal*, *RAD*, and *ContextMap*) did not detect any connections, resulting in a recall of 0 (and the calculation of the precision is not applicable). Two other tools only extracted a miniscule amount of connections (*Prophet* showed four true positives resulting in an F1-score of 0.02; *RAD-source* detected 7 connections, resulting in an F1-score of 0.04) and can hence also be considered as not effective in extracting this characteristic. These results occurred despite us having confirmed that the tools executed correctly.

From the three tools showing better extraction correctness, *MicroDepGraph* and *Code2DFD* showed a high precision (0.98 and 0.92, respectively), and *Code2DFD* and *Attack Graph Generator* a good recall (0.86 and 0.83, respectively). With a value of 0.39, the precision of *Attack Graph Generator* is rather low, and with a value of 0.51, the recall of *MicroDepGraph* as well. These yield a F1-score of 0.87 for *Code2DFD*, 0.67 for *MicroDepGraph*, and 0.54 for *Attack Graph Generator*.

Overall, *MicroDepGraph* shows a high precision but low recall and *Attack Graph Generator* the opposite. *Code2DFD* is more balanced and achieves high scores in both metrics. This results in *Code2DFD* having the highest F1-score.

Answer to RQ4.2: Table 6b presents the tools' observed accuracy in extracting the analyzed applications' connections. The highest F1-score of 0.87 was observed for *Code2DFD*, followed by *MicroDepGraph* (F1 = 0.67) and *Attack Graph Generator* (F1 = 0.54).

Endpoints Six of the evaluated tools extract the applications' REST endpoints during analysis, i.e., endpoints over which RESTful API requests can be made to services. Table 6c shows the results of the six tools for this characteristic. Three tools (*MicroGraal*, *Prophet*, and *ContextMap*) failed in detecting any endpoints completely, resulting in a recall of zero and non-applicable precision. From the other three, we observed a very high precision for *RAD* (0.94) with only six false positives. It also had a good recall with 0.68. *RAD-source* had the second-highest precision with a value of 0.89, but a low recall of 0.54. Similarly, *Code2DFD* showed a high precision (0.87) but low recall, achieving 0.54 in this metric. Combined into the F1-score, *RAD* scores 0.79, *RAD-source* 0.67, and *Code2DFD* 0.66.

Answer to RQ4.3: Table 6c presents the tools' observed accuracy in extracting the analyzed applications' endpoints. *RAD* achieved the highest F1-score of 0.79. *RAD-source* and *Code2DFD* performed comparably well with F1-scores of 0.67 and 0.66, respectively.

All Characteristics in Extraction Scope Combining the results per characteristic presented above, Table 6d presents again each tool's extraction scope and the observed results for all characteristics in its extraction scope. In other words, the table shows each tools' correctness for performing the architecture extraction it is intended to do. Two tools (*MicroGraal* and *ContextMap*) failed completely, and three more tools (*Prophet*, *RAD*, and *RAD-source*) also achieved low scores overall when looking at the F1-score as a combination of precision and recall (scores of 0.10, 0.38, and 0.29, respectively). The other three tools exhibited similar results as for the components and connections: *Attack Graph Generator* showed a high number of true positives but also false positives and a resulting highest recall of 0.86 but low precision; *MicroDepGraph* showed a low number of false positives but also less true positives resulting in the highest precision of 0.99 but low recall; and *Code2DFD* showed more balance between the measures and a resulting good precision, good recall, and overall highest F1-score with some distance of 0.86.

Execution Time The tools' execution times for performing the architecture extraction were measured to allow an assessment of their suitability to be integrated in time-sensitive analysis settings such as fast-paced CI/CD pipelines. Table 7 presents the results.

RAD showed the quickest execution times with an average of 0.03 seconds per application over ten execution runs, directly followed by *RAD-source* with 0.04. *Prophet* and *microMiner* also average well below a second per application, with 0.17 and 0.20, respectively. *Attack Graph Generator* (1.3) and *MicroDepgraph* (1.5) average slightly more than

Table 7 Tools' execution times for analyzing the indicated number of applications individually. Average over ten execution runs. Tool with dashes (—) requires manual steps to a large degree

	Analyzed	Time per App. [sec]		
Tool	Apps.	Min.	Max.	Avg.
Attack Graph Generator	9	1.2	1.5	1.3
Code2DFD	17	6.7	7.8	7.3
MicroDepGraph	10	1.3	1.6	1.5
MicroGraal	10	50	144	60
microMiner	5	0.20	0.22	0.20
Prophet	17	0.16	0.17	0.17
RAD	13	0.03	0.03	0.03
RAD-source	17	0.04	0.04	0.04
Contextmap	10	—	—	—

one second, while code2DFD takes 7.3 seconds per application on average. With an average of one minute per application, *microGraal* showed the longest execution times in our study.

4.2 Combinations of tools

To investigate whether multiple tools could be combined to exceed the extraction correctness of the individual tools, the results of different combinations of tools were combined and the scores they achieved were assessed. Based on the results of the individual tools, only a small subset of all possible permutations of tools are reasonable to investigate.

For the characteristics components and connections, the tools *Attack Graph Generator*, *Code2DFD*, *microMiner*, and *MicroDepGraph* are the only ones that achieved good results. A comparison of the detailed results shows that *microMiner* does not provide any additional value over the others in terms of detecting characteristics that the others did not and is therefore omitted, leaving *Attack Graph Generator*, *Code2DFD*, and *MicroDepGraph* for the tool combinations. For the detection of endpoints, the results of *Code2DFD*, *RAD*, and *RAD-source* are all high and distinct, i.e., they were successful in the detection of different endpoints. The permutations of all three are hence assessed for these characteristics. Finally, for the extraction of all characteristics, we combine the best tool combinations for the detection of each characteristic per metric and compare them against a combination of all seven tools that were part of the study and produced results for any characteristic. Table 8 presents the combinations of tools considered in the evaluation, resulting from the above considerations.

Components Table 9a shows the results for the tool combinations' extraction correctness for the characteristic components. For each metric, there is a combination that achieves a high score. All evaluated "AND"-combinations exhibit a perfect precision of 1.0, whereas combining tools with an "OR" operation yields a high recall of 0.98 for three combinations and 0.87 for $\mathrm{AM}_{\mathrm{OR}}$. High F1-scores are also observed, with the combination $\mathrm{CM}_{\mathrm{OR}}$ yielding the best result of 0.98 in F1-score. Interestingly, the best combination of tools in terms of recall outperforms the best individual tool, *Code2DFD*, by only a single true positive. *Code2DFD* alone matches the F1-score of the best tool combination in this metric.

Connections Table 9b shows the results for the combinations of tools for the characteristic connections. The tool combinations $\mathrm{ACM}_{\mathrm{OR}}$ and $\mathrm{AC}_{\mathrm{OR}}$ exhibit the same results, showing that *MicroDepGraph* does not contribute any additional true positives or false positives over

Table 8 Combinations of tools that were evaluated for each characteristic

		Included Tools						
Characteristics	ID	AGG	C2D	MDG	MMI	PRO	RAD	RAS
Cp, Cn, E	All	■	■	■	■	■	■	■
	Best_P	*Combination with highest P for each characteristic*						
	Best_R	*Combination with highest R for each characteristic*						
	Best_{F1}	*Combination with highest F1 for each characteristic*						
Cp	ACM	■	■	■	□	□	□	□
	AC	■	■	□	□	□	□	□
	AM	■	□	■	□	□	□	□
	CM	□	■	■	□	□	□	□
Cn	ACM	■	■	■	□	□	□	□
	AC	■	■	□	□	□	□	□
	AM	■	□	■	□	□	□	□
	CM	□	■	■	□	□	□	□
E	CRR	□	■	□	□	□	■	■
	CRD	□	■	□	□	□	■	□
	CRS	□	■	□	□	□	□	■
	RDS	□	□	□	□	□	■	■

Cp = components; **Cn** = connections; **E** = endpoints. **AGG** = Attack Graph Generator, **C2D** = Code2DFD, **MDG** = MicroDepGraph, **MMI** = MicroMiner, **PRO** = Prophet, **RAS** = RAS-source.

the other two tools. As was observed for the extraction of components, the tool combinations with "AND" show a perfect precision and the combinations with "OR" a high recall for the extraction of connections. Specifically, AC_OR achieves a recall of 0.97 as the best combination in this metric. With this result, the combination of *Attack Graph Generator* and *Code2DFD* shows a substantial improvement over the recall of the best individual tool alone, *Attack Graph Generator*, with a recall of 0.86. In terms of F1-score, the results are generally lower, mainly caused by many false positives for combinations that contain *Attack Graph Generator* and many false negatives for "AND"-combinations. Consequently, the combination CM_OR shows the highest F1-score of 0.89. It slightly outperforms the best individual tool, *MicroDepGraph*, which scored 0.88 in this metric.

Endpoints Table 9c presents the results for the tool combinations' extraction correctness for the characteristic endpoints. All "AND"-combinations show slightly higher precision than the "OR"-combinations again, with the highest being 0.96 for CRS_AND. The combination of all three considered tools CRR_OR yields the highest recall of 0.88, a substantial improvement over the best individual tool *RAD* (recall of 0.68). The F1-score can be marginally improved to 0.85 over the score of 0.79 for *RAD* alone when combining it with *Code2DFD* into CRD_OR.

All characteristics Table 9d shows the results for the combinations of tools for the full architecture extraction, i.e., components, connections, and endpoints. The combinations Best_P, Best_R, and Best_{F1} consist of the best-performing tool or combination of tools for each characteristic in terms of the metric indicated by the combination's ID. Table 10 shows the tool combinations resulting from this selection, which showed to yield the best possible results for each evaluated metric:

Table 9 Tool combinations' extraction results for the characteristic (a)*components*, (b) *connections*, (c) *endpoints*, and (d) *all characteristics*

(a) Components

Tool Combination	**GT**	**TP**	**FP**	**FN**	**P**	**R**	**F1**
ACM_{OR}	182	179	46	3	0.80	**0.98**	0.88
ACM_{AND}	182	145	0	37	**1.00**	0.80	0.89
AC_{OR}	182	179	46	3	0.80	**0.98**	0.88
AC_{AND}	182	158	0	24	**1.00**	0.87	0.93
AM_{OR}	144	125	41	19	0.75	0.87	0.81
AM_{AND}	144	82	0	62	**1.00**	0.57	0.73
CM_{OR}	182	178	5	4	0.97	0.98	**0.98**
CM_{AND}	182	153	0	29	**1.00**	0.84	0.91

(b) Connections

Tool Combination	**GT**	**TP**	**FP**	**FN**	**P**	**R**	**F1**
ACM_{OR}	385	373	412	12	0.48	**0.97**	0.64
ACM_{AND}	385	208	0	177	**1.00**	0.54	0.70
AC_{OR}	385	373	412	12	0.48	**0.97**	0.64
AC_{AND}	385	279	0	106	**1.00**	0.72	0.84
AM_{OR}	325	282	396	43	0.42	0.87	0.56
AM_{AND}	325	121	0	204	**1.00**	0.37	0.54
CM_{OR}	385	331	29	54	0.92	0.86	**0.89**
CM_{AND}	385	219	0	166	**1.00**	0.57	0.73

(c) Endpoints

Tool Combination	**GT**	**TP**	**FP**	**FN**	**P**	**R**	**F1**
CRR_{OR}	160	141	23	19	0.86	**0.88**	0.87
CRR_{AND}	160	28	2	132	0.93	0.18	0.29
CRD_{OR}	160	132	17	28	0.89	0.83	**0.85**
CRD_{AND}	160	44	2	116	0.96	0.28	0.43
CRS_{OR}	160	120	20	40	0.86	0.75	0.80
CRS_{AND}	160	52	2	108	**0.96**	0.33	0.49
RDS_{OR}	160	123	12	37	0.91	0.77	0.83
RDS_{AND}	160	53	3	107	0.95	0.33	0.49

(d) All characteristics

Tool Combination	**GT**	**TP**	**FP**	**FN**	**P**	**R**	**F1**
All_{OR}	727	694	509	33	0.58	**0.95**	0.72
All_{AND}	727	52	2	675	0.96	0.07	0.13
$Best_{P}$	727	489	2	238	**1.00**	0.76	0.80
$Best_{R}$	727	694	484	33	0.59	**0.95**	0.73
$Best_{F1}$	727	650	57	77	0.92	0.89	**0.91**

GT = number of individual characteristics in the applications that the combination should be able to analyze; **TP**/**FP**/**FN** = observed true positives / false positives / false negatives; **P**/**R**/**F1** = calculated precision / recall / F1-score

Table 10 Optimal tool combinations for each evaluated metric

Combination	Components	Connections	Endpoints
$Best_{P}$	AC_{AND}	AC_{AND}	CRS_{AND}
$Best_{R}$	AC_{OR}	All_{OR}	CRR_{OR}
$Best_{F1}$	C2D	CM_{OR}	CRR_{OR}

All tool combinations in Table 9d consider the full dataset of 727 individual characteristics in the ground truth, i.e., the metrics were not inflated by simply excluding tools that are able to analyze more applications but also produce more FPs. An almost perfect precision of 1.00 after rounding can be achieved with the tool combination Best_P(consisting of *Attack Graph Generator*, *Code2DFD*, and *RAD-source*) while still having a reasonably high F1-score of 0.80. Combining all seven tools that were part of this evaluation with an "AND"-operator also achieves a high precision of 0.96, but a low recall and F1-score. The highest recall of a combination of a subset of tools can not outperform the combination of all tools per definition, but also, no other combination of only a subset of tools achieves the same recall combined with a substantially higher precision. Only a slight improvement can be achieved in this regard. The set of tools in Best_R thus contains all tools also contained in All_OR. Finally, the best possible overall extraction correctness measured in F1-score is 0.91, showing both a high precision and recall. It is a combination of four tools, *Code2DFD*, *MicroDepGraph*, *RAD*, and *RAD-source*. This combination is an improvement of ∼5% over the best individual tool, *Code2DFD*, which showed an F1-score of 0.86.

Answer to RQ5: Combining multiple tools improves the extraction accuracy in almost all investigated cases. For each evaluated performance metric and each characteristic (except of the F1-score for components), a tool combination can be found that outperforms the results of the individual tools. A combination of *Code2DFD*, *MicroDepGraph*, *RAD*, and *RAD-source* achieves the highest F1-score of 0.91.

5 Discussion

The results presented in Section 4 show the identified tools' accuracy in architecture extraction. We discuss the results in the following.

5.1 Tools' extraction accuracy and execution time

As a first, general observation concerning the results of the presented comparison of tools, the extent to which valid results could be obtained is limited. Out of the nine tools considered during the comparison, only six exhibited an extraction correctness that even warrants an inclusion in the comparison concerning at least one characteristic. On the other hand, those tools that did produce useful results show promising extraction correctness, especially when also considering the combinations of individual tools. For all three evaluated metrics (precision, recall, and F1-measure), at least one tool or combination of tools exhibited a value of over 0.90, which we consider to be a generally good score without further context.

It is a known issue of static analysis security testing (SAST) tools, that they often produce a high number of false positive warnings, which is a major factor in inhibiting their adoption by practitioners (Christakis and Bird 2016; Johnson et al. 2013). In the context of this work, falsely detected characteristics could likely have a comparable effect on the user-experience. Some evaluated tools exhibited concerning behavior in this regard, for example,

Attack Graph Generator with more false FPs than TPs in the detection of connections and an overall precision of 0.48. Nevertheless, there are four tools with an overall precision above 0.90 (*Code2DFD*: 0.91, *MicroDepGraph*: 0.99, *microMiner*: 0.97, and *RAD*: 0.98) and three combinations of tools (All_{AND}: 0.97, Best_{P}: 0.99, and Best_{F1}: 0.93).

In some use-cases, a high recall might be more important than the precision. For this metric, the best-performing individual tool *Attack Graph Generator* achieves a value of 0.86 over all characteristics in its extraction scope, which could be considered low depending on the scenario. Here, combining multiple tools proves to be a valuable approach, with the combination Best_{R} consisting of *Attack Graph Generator*, *Code2DFD*, *RAD*, and *RAD-source* exhibiting a recall of 0.96.

Finally, with the F1-score combining precision and recall into a single metric, it can be considered as a suited overall assessment criterion. A value of 0.86 over all characteristics was observed for *Code2DFD* and can be improved to a result of 0.91 when combining *Code2DFD* with *MicroDepGraph*, *RAD*, and *RAD-source*. Whether this is seen as sufficiently correct has to be decided for a specific use-case, however, we see it as a reasonably good result, especially when considering that the tools show potential to be improved (see Section 5.2, lesson learned 3 and 4).

Concerning the tools' execution times, all tools that produced meaningful results showed short execution times per application. *RAD*, *RAD-source*, *Prophet*, and *microMiner* all average well below a second per application, *Attack Graph Generator* and *MicroDepGraph* around 1.5 seconds, and *Code2DFD* averages 7.3 seconds per application. Only *MicroGraal* – which also did not produce any results – takes a full minute per application to run on average. These execution times should not pose any problems for most scenarios. The distribution of the execution times also resembles the complexity of the tools' analysis approaches. *Code2DFD* with the most in-depth approach takes the longest to run, while the tools simply parsing deployment files are much quicker.

Regarding the use of multiple tools in combination over a single tool, the observed results suggest that it is beneficial to do the former in most cases. *Code2DFD* is the only tool that showed extraction correctness that could not be improved by combining it with other tools, but only for the characteristic components and only in F1-score. For the other characteristics and metrics, a combination was found that outperformed the best individual tool for this case.

We note that the reported execution times for combinations of tools are simply the sum of the individual tools' execution times. The numbers could show to be higher in reality because of added overhead due to the integration and, e.g., a voting mechanism, but that they could also be improved if the implementations are truly joint into a single tool combining the multiple analysis techniques. With the observed times as the basis for an assessment, we can say that they should be sufficiently low for a use of the tools in most analysis settings.

5.2 Lessons learned

Some general observations derive from the detailed discussion of the results above. They are presented below as *lessons learned* and could indicate where future research efforts might best be pointed.

Lesson learned 1: Basic architecture extraction can be achieved with high precision and in short time via simple parsing of deployment files Parsing deployment files to extract the analyzed systems' basic architecture (i.e., components and/or connections) is the most common employed technique amongst the tools in our comparison. All but one tool apply this technique: *Attack Graph Generator*, *Code2DFD*, and *MicroDepGraph* parse Docker Compose files, *Code2DFD* also parses Docker files, and *MicroMiner* parses Kubernetes files. In fact, for the comparison of tools, the results of the *Attack Graph Generator* and *microMiner* are solely achieved via this technique. The high precision of *microMiner* in extracting components in this way and the prevalence of the technique amongst the tools is a testament to its effectiveness. Deployment files are simple, unambiguous documents that can be easily and very efficiently parsed, resulting in high precision and very short execution times of the tools relying on this technique. Considering that only a few different containerization and orchestration solutions could cover a large part of microservice applications emphasizes the benefits of this technique. Tools developed in the future should capitalize on this and make a simple parsing of deployment files part of their analysis approach.

Lesson learned 2: Achieving a high recall requires a deeper analysis of the source code beyond deployment files and slightly longer execution times Although deployment files enable tools to achieve high precision with a very simple detection technique, they critically do not capture all relevant system characteristics. Not only are whole groups of system characteristics not always or never detectable via these files (components are the only group that is always included), but also are individual instances of a group not always present. There are a total of 65 components that are not included in a deployment file in the used dataset. Their existence has to be detected deeper in the source code. An approach followed by multiple tools is the identification of Java annotations. They are easily detectable and simultaneously have a high information content concerning the architecture extraction. Other system characteristics can only be detected, e.g., based on the existence of dependencies or plugins. To this end, following heuristics can yield good results, for example by inferring the existence of system characteristics based on the detection of the required import statements. Tool developers should carefully consider which indicators in the source code can be leveraged to achieve a high recall and investigate more complex detection techniques beyond simple parsing. The importance of a deeper code analysis can be seen in the observed results, where only tools that go beyond the deployment files in their analysis achieve high values of recall. This also leads to longer execution times, however, all observed times are still reasonably low and should not pose a problem for the majority of scenarios.

Lesson learned 3: The biggest causes of false positives are inaccurate heuristics and flaws in the tools' implementations A manual assessment tracing the observed false positives to the tools' source code in order to understand their reasons revealed that there are two main causes for them. First, using heuristics to drive the architecture extraction works well in many cases but can also backfire. For example, *Prophet* detects components via the analyzed repository's structure and creates a component for every folder in it. Although many repositories containing microservice applications on GitHub are structured such that each individual microservice is in a folder of its own, this approach also leads to many false positives because not every folder contains the code of a microservice. The second main cause are flaws in the implementation of the tools. These are false positives that are not based on

the underlying approaches of the tools, but result from an improper implementation of the approach. This was especially prevalent for *Code2DFD* with its more complex extraction logic compared to the other tools. Developers of architecture reconstruction tools should put effort into resolving such flaws in the code, since a high number of false positives is a widely known issue in the context of static analysis tools that can impede their adoption.

Lesson learned 4: Improving the extraction accuracy to almost-perfect results seems possible with existing techniques Despite showing shortcomings or complete failures of individual tools, the presented results are promising when looking at the group of tools in whole. Tools or combination of tools are available in the literature that achieve very good results for all evaluated metrics (precision, recall, and F1-score). Further, an examination of all undetected system characteristics showed, that none of the characteristics exhibits any property that would make it infeasible to detect with either of the evaluated tools' techniques. We believe, that the tools' underlying approaches are well-suited to identify all characteristics in the used dataset while keeping the number of false positives low. A more careful implementation of the prototypes is needed for this, as mentioned in lesson learned 3. Specifically, we believe that an approach that features parsers for a small set of different deployment files as a foundation, and enhances it with a deeper source code analysis should be able to exceed the currently observed results. The tool *Code2DFD* already employs parsing of different deployment files and deep source code analysis. A more mature implementation of it than the current prototype combined with the inclusion of detection techniques from *RAD* and *RAD-source* should be able to yield a very high score in extracting all system characteristics across all evaluated metrics, according to our qualitative assessment.

Lesson learned 5: Existing tools' reproducibility is limited Although we did not investigate the property of reproducibility systematically, the descriptions of our experience in executing the tools and the missing results for many tools despite them having executed successfully show issues in this regard. While some tools could be run without hindrance, others required some adjustments or debugging effort or did not at all perform in the way they were presented in their corresponding publications. Although most of the tools in our study are prototypes for academic purposes instead of carefully maintained products, this indicated degradation of reproducibility is concerning and an issue both in the academic sense and for the adoption of the approaches by practitioners. Tool developers should prioritize well-functioning implementations and deployments to not hinder a wider adoption of their tools due to usability issues.

6 Limitations

6.1 Deviations from registered reports

This study was performed in strict accordance to the methodology presented in the registered report where possible. There were, however, two minor, unforeseen deviations necessary which were imposed by the analyzed tools.

First, we had expected that all identified tools would have components and connections in their extraction scopes, and that all could be compared on these two characteristics at

least. As shown in Table 4, only five of the nine tools consider components and eight consider connections in their architecture extraction. Further, only four tools showed results that can be considered somewhat effective for the extraction of components and three tools for the extraction of connections.

Second, we did not extend the dataset with ground truth for all characteristics in the extraction scopes of the identified tools. Specifically, we omitted this step for the characteristics CRUD operations (only considered by *authz-flow-analysis*), attack graphs (only considered by *Attack Graph Generator*), and access control inconsistencies (only considered by *MicroGraal* and *Prophet*). There are multiple reasons for the exclusion: for characteristics that are only considered by a single tool, a comparison of extraction accuracy would not have been possible; the extraction of access control inconsistencies in the tools is not fully automated but follows a human-in-the-loop approach; and the tool considering CRUD operations had to be excluded from the comparison.

Finally, a new research question has been added compared to the registered report (RQ5, concerning the combinations of tools). We believe it to be a valuable addition to the presented results, and there is no impact on any other parts of the study. The methodology is a natural extension of the one that has been peer-reviewed, and the sections that were not part of the registered report are clearly marked. Therefore, we see no introduced threats to validity or malpractice concerning the process of registered reports in this.

6.2 Anticipated risks

In our registered report, we had anticipated three risks that could have occurred during the conduction of the planned study. Now, after having executed all steps, we revisit these risks in the following.

Risk 1: Inability to Execute Tools. While some tools posed difficulties in running them, we received valuable support from all authors we had contacted and were then successful in executing most of them. However, one tool (*Prophet2*) could not be considered further in the study due to our inability to run it. A detailed description of the problem is given in Section 3.3. In short, the tool has been abandoned since its publication, we were not able to fix the tool on our own, and the authors have moved on from academia.

From those tools that were part of the comparison, two were not successful in producing any results (*MicroGraal* and *ContextMap*) and others extracted only few individual characteristics from their extraction scope (*Prophet*, *RAD*, *RAD-source*). However, all tools indicated a successful execution despite producing no or very limited results and are thus seen as valid subjects.

Risk 2: Tools' Extraction Scopes too Distinct for Comparison. For the case that the identified tools have extraction scopes that are distinct from each other and thus not allow any comparison of extracted characteristics, we had planned to compare them on the characteristics components and connections, as we believed these to be in the extraction scope of every architecture recovery tool. The study showed, that this assumption is not true for all tools (see Table 4). Some require the list of components to be provided, and one does not consider connections. Nevertheless, the three evaluated characteristics are in the extraction scopes of five or more tools, thus providing a valuable comparison.

Across all identified tools, there are three additional system characteristics that are only extracted by a single tool or two tools each (CRUD operations by *authz-flow-analysis*,

which was excluded anyway; attack graphs by *Attack Graph Generator*; and access control inconsistencies by *MicroGraal* and *Prophet*). The anticipated risk hence did materialize, as the tools could not be compared concerning the extraction accuracy for these additional characteristics.

Risk 3: Characteristics not existent in Dataset. The used dataset was extended with full information on endpoints in the contained applications, as described in Section 2.3. Three additional system characteristics are in the extraction scope of one or two of the identified tools, but not existent in the ground truth dataset. Additionally, for two of them, it is unclear how the ground truth could be created manually. We therefore decided not to extend the dataset with them, as no comparison would have been possible.

One other case regarding this anticipated risk is the tool *protoc-gen-scip*, which was identified in the literature but excluded from the comparison. The reason is described in Section 3.3. In summary, it is made for microservice applications using the RPC protocol. The authors use one open-source application and one application made specifically for this use to evaluate their tool, and they mention the scarcity of applications using the RPC protocol. Extending the dataset with new applications would have meant a considerable effort without a fair comparison of tools, since they would have been executed on different applications. For this reason, this tool was not considered further in the study.

6.3 Threats to Validity

The work in this paper is subject to some threats to validity that could have influenced the process and hence limit the presented results' validity. They are presented in the following together with employed mitigations. We recall that this paper's methodology has undergone peer-review before and was published as a registered report prior to the execution of the presented work.

Internal Validity The authors' subjective judgement is needed at multiple places of a literature review. The identification of tools via a multivocal literature review in this paper could therefore have been influenced by selection bias, search bias, and extraction errors during the selection and examination of sources. We addressed these issues by following robust and established methods for performing literature reviews. Specifically, all steps relying on subjective decisions were performed by two authors independently and conflicts solved with a third author, inter-rater agreement was measured and reported in terms of Cohen's Kappa, reproducible selection criteria and details on identifying tools in the selected sources were reported, and a replication package containing all data needed to evaluate the validity of the intermediary and final results is made available. Despite these efforts, we might not have been successful in identifying all tools in the targeted scope of this work, e.g., because of publication bias which could prevent the publication of tools that follow an approach that is not sufficiently novel to be accepted in the academic literature. The consideration of the gray literature in addition to academic sources is designed to mitigate this threat to validity, but could in turn be subject to database bias, i.e., could have been affected by the choice of gray literature sources that were analyzed. The inclusion of broad sources (especially Google Search) was precisely intended to mitigate the above-described threat.

The quality of the gray literature sources was not evaluated or taken into consideration in the selection process, as is done in some gray literature reviews because these sources are not peer-reviewed per definition. We decided against such a process because the existence of a reference to a tool that fits the scope of this study is not affected by the quality of the source.

The measurement of the tools' correctness in architecture recovery relied on manual work, both in the creation of the ground truth and in the analysis of the outputs. However, the reliability of the used dataset's correctness is strengthened by the fact that the initial dataset has been published and peer-reviewed before, and that the extensions made in the context of this work are minor and were obtained by repeating activities in line with those of the original, peer-reviewed process. The correctness of translating the tools' outputs into the chosen measurements and metrics is made more reliable by involving two authors in the process and by providing a replication package supporting all results.

Evaluating the tools' correctness was only possible for tools that could be executed successfully and that produced meaningful results. One tool had to be excluded from the study based on our inability to run it, and two tools were executed and indicated a successful analysis but did not produce any results. We spent considerable effort on these cases and sought advice from the authors, but ultimately did not succeed. Although this may limit the expressiveness of the presented results, we believe that no further feasible actions could have been taken.

External validity The presented results and drawn conclusions might not entirely map to other execution scenarios. All evaluated tools are either specific to Java or independent of the analyzed applications' programming language, which matches the applications in the used dataset. Nevertheless, the dataset is the biggest factor that could have influenced the observed results' generalizability, since the applications are small- to medium-sized and show relatively homogeneous architectural patterns. Thus, the tools could perform differently on bigger or more complex applications, and other conclusions could have been drawn concerning their comparison. However, the dataset is the largest one currently available in the literature for this purpose. Future work could include the creation of a dataset containing more industry-near applications, or a replication of the work if such a dataset is published by others.

Some identified tools extract more characteristics than the ones evaluated in the comparison, as reported. Their full functionality for architecture recovery is hence not covered by the comparison, only the described characteristics are.

Construct validity The tools were evaluated based on precision, recall, and F1-measure, which are commonly used and objective metrics for this use. They capture the tools' most important requirement, their correctness in architecture recovery. Additionally, the tools' execution times were measured to provide a basis for assessing their suitability for being used in time-sensitive contexts. Nevertheless, other metrics might be more important for specific use-cases, and the presented results could show a different performance when evaluated on these.

Most of the tools were initially presented as prototypes representing an analysis approach. Therefore, the results in this paper can not be seen as evaluation of the approaches themselves, because the implementations might contain flaws and thus not accurately represent the intended approach. Since this paper aims to provide a comparison of available tools and not approaches, this does not limit the validity of the drawn conclusions.

Finally, some of the identified tools follow a hybrid approach for architecture reconstruction, where an additional dynamic analysis follows the static creation of the architecture representation. Our omission of the dynamic part of the analysis and restriction to the static part could potentially fail to accurately represent the tools' capabilities. We mitigated this limitation by explicitly describing such cases in the presentation of the identified tools and by considering only hybrid tools where the dynamic analysis does not contribute an integral part to the architecture reconstruction according to the corresponding paper, but merely adds further information that is not relevant for our study.

7 Related work

Various secondary studies have been presented in the realm of microservice architecture. Most early work in this regard focused on delineating the similarities and differences between the microservice architecture and the monolithic architecture, service-oriented architecture, and other prior styles, as well as on outlining the trends, benefits, and downsides of the microservice architecture (e.g., Dragoni et al. (2013) with their "yesterday, today, and tomorrow" of microservices or Soldani et al. (2018) with their "pains and gains" of using the microservice architecture).

Other work considered the provided tool support of presented techniques early-on already. In 2016 (two years after the emergence of the term "microservice architecture" following Lewis and Fowler (2014)), Pahl and Jamshidi conducted a systematic mapping study (Pahl and Jamshidi 2016) to review the existing literature on the microservice architecture at the time. One of the considered aspects in their work was the tool support available for software engineering activities related to microservices. The authors identified two tools for the migration of monolithic applications to the microservice architecture, but they note a general lack of tool support at the time of publishing their work. Similarly, Alshuqayran et al. (2016) conducted a systematic mapping study on the microservice architecture in the same year, identifying trends that were visible in the academic literature at the time. They identified a wide variety of modeling diagrams being used to represent the architecture of microservice applications, but no work on their recovery.

The usefulness of architecture recovery techniques for different use-cases have been noted by some authors. For example, Abdelfattah and Cerny (2023) conducted a rapid review of the literature concerning reasoning in microservices – a term covering assessment and understanding of an existing system, according to the authors – and evolution of microservice systems. They state architecture recovery to be at the core of the reasoning process and emphasize its necessity to obtain a holistic view of microservice applications. Their description of analyzing the evolution of microservice applications is also based on a recovered architectural view. Multiple approaches for architecture recovery are identified in the paper, some of which also provide implementations. However, the existence of tool support is not a focus of the review. Bushong et al. (2021) performed a systematic map-

ping study on approaches for microservice analysis and architecture evolution. Multiple of the techniques they identified have architecture reconstruction as their main goal and others as a means to achieve other goals such as analyzing security or system evolution. One of the dimensions the authors used to classify the findings of their study is whether a tool is provided for proposed approaches. Eleven tools were identified by the authors of the study across 55 presented papers. One tool performs architecture reconstruction, however, dynamically (as was pointed out by Bakhtin et al. (2024)), and is therefore not in the scope of our study.

Most of the related work on microservice applications and their analysis present work on the same abstraction level as the architecture recovery tools targeted by our study create. Although it is not explicitly mentioned in all cases, many of these findings arguably require an architecture recovery tool as a preliminary. For example, both Neri et al. (2020) and Ponce et al. (2022) conducted multivocal literature reviews on smells of microservice applications. The former is focussed on architectural smells directly, while the latter is more security-oriented, and some of the identified security smells are at the architectural level as well.

The availability of tool support for identified approaches is often reported in literature reviews for other software engineering activities for microservice applications. For example, Fritzsch et al. (2019) conducted a review of microservice refactoring approaches for the migration of applications from a monolithic to a microservice architecture. The authors identified prototype implementations for three and a more mature tool for one of the ten compared approaches. Saucedo et al. (2024) also targeted the migration of monoliths to microservices with a systematic mapping study, but are more generally concerned with depicting the typical activities of this process. They identified 31 tools that are used in refactoring activities, including general utilities such as tools for source control. Gortney et al. (2022) present the findings of a systematic mapping analysis of architecture visualization approaches that perform dynamic analysis. From the 20 identified approaches, 13 are tool-supported. A systematic literature review by Lelovic et al. (2024) found 19 tools for change impact analysis in microservice applications. Bakhtin et al. (2022) performed a gray literature review of tools detecting API patterns, finding 59 tools. A gray literature review presented by Giamattei et al. (2024) identified 71 tools for monitoring microservice applications. Work closely related to our study was published by Cerny et al. (2022), who conducted a review of static and dynamic microservice architecture reconstruction approaches and visualization techniques. Finally, the systematic mapping study by Bakhtin et al. (2024) that this paper is based on also identified tools for architecture recovery. All these reviews, however, present the identified techniques and tools entirely based on information reported in the corresponding publications. None of them executed the found tools and report on the observations, as we did in this paper.

Studies comparing tools based on results observed when executing them have been presented in the literature for other tasks than architecture recovery and other domains than microservices. For example, this general study methodology has been used to compare static analysis tools for quality assurance by Mantere et al. (2009), by Lenarduzzi et al. (2023), by Li et al. (2023), and by Liu et al. (2023) or to compare bug detection tools by Habib and Pradel (2018), by Tomassi (2018), and by Thung et al. (2015). Even comparisons of tools for software architecture recovery have been presented, e.g., by Lutellier et al. (2015, 2018) and by Garcia et al. (2013). Although the above studies base their findings on executing the

compared tools, as we did as well, they concern different domains than the microservice architecture.

Finally, two recent publications also follow the approach of executing identified tools to compare them, and they both target the microservice architecture domain. First, Akkaya and Ovatman (2022) identified three tools for microservice decomposition – i.e., tools for the migration of monolithic applications to the microservice architecture – from multiple prior literature reviews and executed them on the same set of applications. Then, they compared the results against a manual decomposition of the applications. Second, Wang et al. (2024) selected eight microservice decomposition tools from the results of a literature review and executed them on a set of applications. The authors then compare the tools based on the observed results concerning several relevant metrics. Although these studies target tools for a different use-case than we did in our work, both also identified the need for a comparison of tools based on observed results on a common dataset instead of reported results and specifications.

In conclusion, to the best of our knowledge, the presented work is the first review of static architecture recovery tools for microservice applications that provides a comprehensive comparison of the tools' performance on a common dataset.

8 Conclusion

This paper presented a comparison of static analysis architecture recovery tools for microservice applications based on their results observed when executing them on a common dataset. A previous literature review by Bakhtin et al. (2024) has been repeated and extended into a multivocal literature review. It resulted in the identification of 13 such architecture recovery tools, which have been characterized concerning the characteristics that they are intended to extract from analyzed applications (their *extraction scope*).

When trying to run the tools, we faced various obstacles for many of them. While no sound methodology for evaluating their usability was followed, our general experience in this step showed a concerning impression of their ease-of-use and the reproducibility of reported results. Nine of the identified tools could successfully be run and were each executed on 17 microservice applications from the microSecEnD dataset (Schneider et al. 2023). The dataset contains dataflow diagrams which serve as ground truth for evaluating the tools' extraction accuracy for the characteristics components, connections, and endpoints. The DFDs in the dataset have been extended with model items for endpoints for this purpose. By comparing the tools' created outputs to the ground truth, we calculated precision, recall, and F1-score of each tool as measures of their correctness in architecture recovery. For each of the three considered characteristics, the results of all tools that have it in their extraction scope were compared. The best-performing tools in terms of precision, recall, and F1-measure over all characteristics in their extraction scope are *MicroDepGraph* (Rahman et al. 2019) ($P = 0.99$), *Attack Graph Generator* (Ibrahim et al. 2019) ($R = 0.86$), and *Code2DFD* (Schneider and Scandariato 2023) ($F1 = 0.86$), respectively.

We also investigated whether it is possible to achieve a higher score in each metric by combining the results of multiple tools. To this end, the results of different combinations of tools were merged with logical "OR" and "AND" operations on the individual characteristic level. The results show, that for each metric and each characteristic, a tool

combination can be found that outperforms or at least matches the accuracy of the best individual tool. Specifically, a combination of *Code2DFD* (Schneider and Scandariato 2023), *MicroDepGraph* (Rahman et al. 2019), *RAD* (Das et al. 2021), and *RAD-source* (Das et al. 2021) showed the highest F1-score (F1 = 0.91) of all evaluated combinations over all characteristics.

The presented work depicts the current state-of-the-art in the domain and can be a valuable resource for researchers and practitioners alike. We summarize our insights and findings in five *lessons learned*, which point out effective techniques and current obstacles, and give pointers for improving static analysis architecture recovery tools for microservice applications in future work.

Acknowledgements We would like to express our gratitude to those authors we contacted for help in running their tools and who all answered supportively.

Author Contributions **Simon Schneider:** Conceptualization, Methodology, Software, Validation, Formal analysis, Data Curation, Writing - Original Draft, Writing - Review & Editing, Visualization, Project administration. **Alexander Bakhtin:** Conceptualization, Methodology, Software, Validation, Formal analysis, Data Curation, Writing - Original Draft, Writing - Review & Editing, Project administration. **Xiaozhou Li:** Conceptualization, Methodology, Writing - Review & Editing. **Jacopo Soldani:** Conceptualization, Methodology, Writing - Review & Editing. **Antonio Brogi:** Conceptualization, Methodology, Writing - Review & Editing. **Tomas Cerny:** Conceptualization, Methodology, Validation, Writing - Review & Editing. **Riccardo Scandariato:** Conceptualization, Methodology, Validation, Resources, Writing - Review & Editing, Supervision, Funding acquisition. **Davide Taibi:** Conceptualization, Methodology, Validation, Resources, Writing - Review & Editing, Supervision, Funding acquisition, Project administration.

Funding Open Access funding enabled and organized by Projekt DEAL. This material is based upon work supported by grants from the Research Council of Finland (grants n. 349487 and 349488 - MuFAno) and from Business Finland (6GSoft project).

Data Availability A replication package of all relevant data is available at Zenodo (2025) and GitHub (https://github.com/M3SOulu/EMSE2025SAR-Replication).

Declarations

Ethical Approval Not applicable.

Informed Consent Not applicable.

Conflicts of Interest The authors declare that they have no conflict of interest.

Clinical Trial Number Not applicable.

References

Abdelfattah AS, Cerny T (1838) Roadmap to reasoning in microservice systems: A rapid review. Appl Sci 13(3):2023

Abdelfattah A, Schiewe M, Curtis J, Cerny T, Song E (2023) Towards security-aware microservices: On extracting endpoint data access operations to determine access rights. In: 13th international conference on cloud computing and services science (CLOSER 2023)

Akkaya K, Ovatman T (2022) A comparative study of meta-data-based microservice extraction tools. IJSSMET

Alshuqayran N, Ali N, Evans R (2016) A systematic mapping study in microservice architecture. In: 2016 IEEE 9th international conference on service-oriented computing and applications (SOCA). IEEE, pp. 44–51

Alshuqayran N, Ali N, Evans R (2018) Towards micro service architecture recovery: An empirical study. In: ICSA

Arisholm E, Briand LC, Hove SE, Labiche Y (2006) The impact of uml documentation on software maintenance: an experimental evaluation. TSE 32(6):365–381

Bakhtin A, Al Maruf A, Cerny T, Taibi D (2022) Survey on tools and techniques detecting microservice api patterns. In: SCC

Bakhtin A, Li X, Soldani J, Brogi A, Cerny T, Taibi D (2024) Tools reconstructing microservice architecture: A systematic mapping study. In: Software Architecture. ECSA 2023 Tracks, Workshops, and Doctoral Symposium, B. Tekinerdoğan, R. Spalazzese, H. Sözer, S. Bonfanti, and D. Weyns, Eds. Cham: Springer Nature Switzerland, pp. 3–18

Bambhore Tukaram A, Schneider S, Díaz Ferreyra NE, Simhandl G, Zdun U, Scandariato R (2022) Towards a security benchmark for the architectural design of microservice applications. In: ARES. New York, NY, USA: ACM

Budgen D, Burn AJ, Brereton P, Kitchenham AB, Pretorius R (2011) Empirical evidence about the uml: a systematic literature review. Software: Practice and Experience 41(4): 363-392

Bushong V, Abdelfattah AS, Maruf AA, Das D, Lehman A, Jaroszewski E, Coffey M, Cerny T, Frajtak K, Tisnovsky P, Bures M (2021) On microservice analysis and architecture evolution: A systematic mapping study. Appl Sci 11(17):7856

Bushong V, Das D, Al Maruf A, Cerny T (2021) Using static analysis to address microservice architecture reconstruction. In: ASE

Cao C, Schneider S, Diaz Ferreyra N, Verweer S, Panichella A, Scandariato R (2024) Catma: Conformance analysis tool for microservice applications. In: ICSE-Companion

Cerny T, Abdelfattah AS, Bushong V, Al Maruf A, Taibi D (2022) Microservice architecture reconstruction and visualization techniques: A review. In: SOSE

Christakis M, Bird C (2016) What developers want and need from program analysis: an empirical study. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, p. 332–343

Das D, Walker A, Bushong V, Svacina J, Cerny T, Matyas V (2021) On automated RBAC assessment by constructing a centralized perspective for microservice mesh. PeerJ Computer Science 7:e376

Di Francesco P, Lago P, Malavolta I (2019) Architecting with microservices: A systematic mapping study. JSS 150:77–97

Do LNQ, Wright JR, Ali K (2022) Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. IEEE Trans Software Eng 48(3):835–847

Dragoni N, Giallorenzo S, Lluch-Lafuente A, Mazzara M, Montesi F, Mustafin R, Safina L (2016) Microservices: yesterday, today, and tomorrow. Springer International Publishing, Berlin

Emam KE (1999) Benchmarking kappa: Interrater agreement in software processassessments. EMSE

Fang A, Zhou R, Tang X, He P (2023) Rpcover: Recovering grpc dependency in multilingual projects. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2023, pp. 1930–1939

Fritzsch J, Bogner J, Zimmermann A, Wagner S (2019) From monolith to microservices: A classification of refactoring approaches. In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Cham: Springer International Publishing

Garcia J, Ivkovic I, Medvidovic N (2013) A comparative analysis of software architecture recovery techniques. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 486–496

Garousi V, Felderer M, Mäntylä MV (2019) Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. IST 106:101–121

Giamattei L, Guerriero A, Pietrantuono R, Russo S, Malavolta I, Islam T, Dînga M, Koziolek A, Singh S, Armbruster M, Gutierrez-Martinez J, Caro-Alvaro S, Rodriguez D, Weber S, Henss J, Vogelin EF, Panojo FS (2024) Monitoring tools for devops and microservices: A systematic grey literature review. JSS

Gortney ME, Harris PE, Cerny T, Maruf AA, Bures M, Taibi D, Tisnovsky P (2022) Visualizing microservice architecture in the dynamic perspective: A systematic mapping study. IEEE Access

Granchelli G, Cardarelli M, Di Francesco P, Malavolta I, Iovino L, Di Salle A (2017) Towards recovering the software architecture of microservice-based systems. In: ICSAW

Gravino C, Scanniello G, Tortora G (2015) Source-code comprehension tasks supported by uml design models: Results from a controlled experiment and a differentiated replication. Journal of Visual Languages & Computing 28:23–38

Gravino C, Tortora G, Scanniello G (2010) An empirical investigation on the relation between analysis models and source code comprehension. In: SAC. ACM

Habib A, Pradel M (2018) How many of all bugs do we find? a study of static bug detectors. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, p. 317–328

Hutcheson R, Blanchard A, Lambaria N, Hale J, David Kozak, AE, Cerny T (2024) Software architecture reconstruction for microservice systems using static analysis via graalvm native image. In: SANER 2024, ser. SANER. Institute of Electrical and Electronics Engineers

Ibrahim A, Bozhinoski S, Pretschner A (2019) Attack graph generation for microservice architecture. In: Symposium on Applied Computing. ACM

JetBrains (2022) The state of developer ecosystem 2022," JetBrains, Tech. Rep. Sccessed on 09.02.2024. [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2022/microservices/

Johnson B, Song Y, Murphy-Hill E, Bowdidge R (2013) Why don't software developers use static analysis tools to find bugs?. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 672–681

JRebel (2022) 2022 java developer productivity report. JRebel, Tech. Rep. accessed on 09.02.2024. [Online]. Available: https://www.jrebel.com/resources/java-developer-productivity-report-2022

Kitchenham B (2004) Procedures for performing systematic reviews. Keele Univ., vol, Keele, UK, p 33

Kitchenham B, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering vol. 2

Kleehaus M, UludagÖ, Schäfer P, Matthes F (2018) Microlyze: A framework for recovering the software architecture in microservice-based environments," in *Information Systems in the Big Data Era*. Springer International Publishing

Landis JR, Koch GG (1977) The measurement of observer agreement for categorical data. Biometrics 33(1):159–174

Lelovic L, Huzinga A, Goulis G, Kaur A, Boone R, Muzrapov U, Abdelfattah AS, Cerny T (2024) Change impact analysis in microservice systems: A systematic literature review. Journal of Systems and Software, p. 112241

Lenarduzzi V, Pecorelli F, Saarimaki N, Lujan S, Palomba (2023) F A critical comparison on six static analysis tools: Detection, agreement, and precision. JSS

Lewis J, Fowler M (2014) Microservices: a definition of this new architectural term. [Online]. Available: https://martinfowler.com/articles/microservices.html

Li K, Chen S, Fan L, Feng R, Liu H, Liu C, Liu Y, Chen Y (2023) "Comparison and evaluation on static application security testing (sast) tools for java," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, p. 921–933

Liu H, Chen S, Feng R, Liu C, Li K, Xu Z, Nie L, Liu Y, Chen Y. (2023) "A comprehensive study on quality assurance tools for java. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, p. 285–297

Lutellier T, Chollak D, Garcia J, Tan L, Rayside D, Medvidovic N, Kroeger R (2015) Comparing software architecture recovery techniques using accurate dependencies. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, pp. 69–78

Lutellier T, Chollak D, Garcia J, Tan L, Rayside D, Medvidović N, Kroeger R (2018) Measuring the impact of code dependencies on software architecture recovery techniques. IEEE Trans Software Eng 44(2):159–181

Mantere M, Uusitalo I, Roning J (2009) Comparison of static code analysis tools. In: SECURWARE

Moreschini S, Recupito G, Lenarduzzi V, Palomba F, Hästbacka D, Taibi D (2023) Toward end-to-end mlops tools map: A preliminary study based on a multivocal literature review ArXiv

Muntoni G, Soldani J, Brogi A (2021) Mining the architecture of microservice-based applications from their kubernetes deployment. In: Advances in Service-Oriented and Cloud Computing. Cham: Springer International Publishing

Neri D, Soldani J, Zimmermann O, Brogi A (2020) Design principles, architectural smells and refactorings for microservices: a multivocal review. SICS

Pahl C, Jamshidi P (2016) Microservices: A systematic mapping study. CLOSER 1:137–146

Peltonen S, Mezzalira L, Taibi D (2021) Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review. IST 136:106571

Ponce F, Soldani J, Astudillo H, Brogi A (2022) Smells and refactorings for microservices security: A multivocal literature review. JSS 192:111393

Quéval P-J, Zdun U (2023) Extracting the architecture of microservices: An approach for explainability and traceability. In: ECSA. Cham: Springer Nature Switzerland

Rahman MI, Panichella S, Taibi D (2019) A curated dataset of microservices-based systems

Ralph P, Ali NB, Baltes S, Bianculli D, Diaz J, Dittrich Y, Ernst N, Felderer M, Feldt R, Filieri A et al (2020) Empirical standards for software engineering research. arXiv:2010.03525

Saucedo AM, Rodríguez G, Rocha FG, dos Santos RP (2024) Migration of monolithic systems to microservices: A systematic mapping study. Inf Softw Technol 177:107590

Schiewe M, Curtis J, Bushong V, Cerny T (2022) Advancing static code analysis with language-agnostic component identification. IEEE Access 10:30 743–30 761

Schneider S, Bakhtin A, Li X, Soldani J, Brogi A, Cerny T, Scandariato R, Taibi D (2024) Comparison of static analysis architecture recovery tools for microservice applications. [Online]. Available: https://arxiv.org/abs/2403.06941

Schneider S, Diaz Ferreyra NE, Queval P-J, Simhandl G, Zdun U, Scandariato R (2024) How dataflow diagrams impact software security analysis: an empirical experiment. In: SANER

Schneider S, Özen T, Chen M, Scandariato R (2023) microsecend: A dataset of security-enriched dataflow diagrams for microservice applications. In: MSR

Schneider S, Scandariato R (2023) Automatic extraction of security-rich dataflow diagrams for microservice applications written in java. JSS

Soldani J, Khalili J, Brogi A (2023) Offline mining of microservice-based architectures (extended version). SN Comput. Sci. 4(3):304

Soldani J, Muntoni G, Neri D, Brogi A (2021) The mtosca toolchain: Mining, analyzing, and refactoring microservice-based architectures. Practice and Experience, Software

Soldani J, Tamburri DA, Van Den Heuvel W-J (2018) The pains and gains of microservices: A systematic grey literature review. JSS 146:215–232

Taibi D, Lenarduzzi V, Pahl C (2020) Microservices Anti-patterns: A Taxonomy Cham: Springer International Publishing

Thung F, Lucia, Lo D, Jiang L, Rahman F, Devanbu PT (2015) To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools. Automated Software Engineering, vol. 22, no. 4, pp. 561–602. [Online]. Available: https://doi.org/10.1007/s10515-014-0169-8

Tomassi DA (2018) Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, p. 980–982

Vassallo C, Panichella S, Palomba F, Proksch S, Gall HC, Zaidman A (2020) "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1419–1457, Mar. [Online]. Available: https://doi.org/10.1007/s10664-019-09750-5

Wang Y, Bornais S, Rubin J (2024) Microservice decomposition techniques: An independent tool comparison. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, pp. 1295–1307

Wohlin C (2014) Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: EASE. ACM

Zenodo (2025) Replication package of this article. [Online]. Available: https://doi.org/10.5281/zenodo.14179612

Simon Schneider is a Ph.D. student at the Institute of Software Security at the Hamburg University of Technology (TUHH), Germany. He works on automated approaches for the detection of security features in source code, architecture reconstruction, and automated architectural security analysis for microservice applications and AI-based systems.

Alexander Bakhtin Alexander Bakhtin is a doctoral researcher at the University of Oulu, Finland. He works in the unit of Empirical Software Engineering in Software, Systems, and Services at the Information Technology and Electrical Engineering Faculty under the supervision of Prof. Davide Taibi. He received his Bachelor's Degree in Science and Technology (mathematics) and Master's Degree in Computing Sciences (machine Learning) from Tampere University, Finland, in 2021 and 2022, respectively. His current research interests include applying static and temporal network methods to networks encountered in the field of software engineering, such as networks of microservice architecture or developer collaboration.

Xiaozhou Li is a postdoctoral researcher (RTDa) in the Faculty of Engineering at Free University of Bozen-Bolzano, Italy. He was previously working as postdoctoral researcher in Empirical Software Engineering in Software, Systems and Services (M3S) Research Group at Faculty of Information Technology and Electrical Engineering (ITEE), University of Oulu, Finland. Li received his Ph.D. in computer science from Tampere University, Finland. His research interests include microservice degradation, microservice organizational structure, open-source software quality, software maintenance and evolution, user review opinion mining, and computational game studies.

Jacopo Soldani is an Associate Professor at the University of Pisa, where he earned a PhD in Computer Science (2017). His research focuses on service-based software engineering–covering cloud-native applications, microservices, and multi-service architectures–while also exploring systematic methods to assess the state-of-the-art and state-of-practice in the field. Jacopo has participated in various research projects, currently serving as Principal Investigator of a national project on Cloud-IoT software sustainability. He contributes to several international software engineering journals, including serving as Associate Editor for the Journal of Systems and Software. Jacopo has also played key roles in major software engineering conferences, including General Chair of IEEE SOSE 2025 and Program Chair of ECSA 2025, IEEE SOSE 2024–2025, and ESOCC 2023.

Antonio Brogi is full professor at the Department of Computer Science, University of Pisa (Italy) since 2004, where he leads the Service-Oriented, Cloud and Fog Computing research group. Since November 2020 he is also the Coordinator of the Ph.D. Program in Computer Science at the University of Pisa, Florence and Siena. His research interests include software engineering, symbolic artificial intelligence, Cloud-Edge computing, sustainability and ICT, and quantum software engineering. He has published his research results in around 300 papers in international journals and conferences.

Tomas Cerny is an Associate Professor of Systems and Industrial Engineering at the University of Arizona, Tucson. After earning Engineering and Masters's degrees from the Czech Technical University, FEE, and from Baylor University, he has served as an Assistant professor at the Science and Computer Department at the Czech Technical University, FEE since 2009. Soon after earning a Doctoral degree in 2016, he returned to Baylor University to join the Computer Science department. He was tenured in 2023 at Baylor and moved as Associate Professor of Systems and Industrial Engineering at the University of Arizona. His research focus is Software Engineering, Static Analysis, Cloud Computing Applications nd Architecture Degradation. He served 15+ years as the lead developer of the International Collegiate Programming Contest Management System. He authored nearly 200 publications, mostly relating to code analysis and aspect-oriented programming. Among his awards are the seven best papers, the 2023 Baylor Scholarship Award, the Outstanding Service Award ACM SIGAPP 2018 and 2015, and the 2011 ICPC Joseph S. DeBlasi Outstanding Contribution Award. He actively serves the scientific community and was on the organizing committee for IEEE SOSE, ESOCC, SANER, ACM SAC, ACM RACS, and ICITCS.

Riccardo Scandariato is a full professor at the Hamburg University of Technology (TUHH) in Germany, where he leads the Institute of Software Security. He has more than 15 years of experience developing innovative methods to identify security flaws. His research focuses on software vulnerabilities, in particular on the use of generative AI in order to localize, repair, and avoid security bugs, both in application code and in Infrastructure-as-Code scripts. More info at: https://scandariato.org

Davide Taibi is a Full Professor at the University of Oulu, where he leads the M3S Cloud research group. His research focuses on cloud-native systems, particularly on identifying and mitigating architectural debt, with a strong emphasis on migrating from monolithic to cloud-native applications. He investigates processes and techniques for developing cloud-native applications, identifying patterns and anti-patterns that impact software quality and maintainability. Beyond academia, Davide actively supports companies in modernizing their software architectures, helping them transition to cloud-native technologies while ensuring software maintainability. He also works with local businesses to implement continuous quality monitoring techniques and address DevOps antipatterns. He served as General Chair for IEEE SANER 2024, ECSA 2026, and ICSA 2027, and as Program Chair for PROFES 2023 and Euromicro/SEAA 2025. Learn more at www.taibi.it

Authors and Affiliations

Simon Schneider[1] · Alexander Bakhtin[2] · Xiaozhou Li[3] · Jacopo Soldani[4] · Antonio Brogi[4] · Tomas Cerny[5] · Riccardo Scandariato[1] · Davide Taibi[2]

✉ Simon Schneider
simon.schneider@tuhh.de

Alexander Bakhtin
alexander.bakhtin@oulu.fi

Xiaozhou Li
xiaozhou.li@unibz.it

Jacopo Soldani
jacopo.soldani@unipi.it

Antonio Brogi
antonio.brogi@unipi.it

Tomas Cerny
tcerny@arizona.edu

Riccardo Scandariato
riccardo.scandariato@tuhh.de

Davide Taibi
davide.taibi@oulu.fi

[1] Hamburg University of Technology, Hamburg, Germany

[2] University of Oulu, Oulu, Finland

[3] University of Oulu and Free University of Bozen-Bolzano, Oulu, Finland

[4] University of Pisa, Pisa, Italy

[5] University of Arizona, Tucson, AZ, USA