Claim: a Lightweight Approach to Identify Microservices in Dockerized Environments

Kevin Maggi
University of Florence
Florence, Italy
kevin.maggi@unifi.it

Roberto Verdecchia
University of Florence
Florence, Italy
roberto.verdecchia@unifi.it

Leonardo Scommegna
University of Florence
Florence, Italy
leonardo.scommegna@unifi.it

Enrico Vicario
University of Florence
Florence, Italy
enrico.vicario@unifi.it

ABSTRACT

Background: Over the past decade, microservices have surged in popularity within software engineering. From a research viewpoint, mining studies are frequently employed to assess the evolution of diverse microservice properties. Despite the growing need, a validated static method to swiftly identify microservices seems to be currently missing in the literature.
Aims: We present Claim, a lightweight static approach that analyzes configuration files to identify microservices in Dockerized environments, specifically designed with mining studies in mind.
Method: To validate Claim, we conduct an empirical experiment comprising 20 repositories, 160 microservices, and 13k commits. *A priori* and manually defined ground truths are used to evaluate Claim's microservice identification effectiveness and efficiency.
Results: Claim detects microservices with an accuracy of 82.0%, reports a median execution time of 61ms per commit, and requires in the worst case scenario 125.5s to analyze the history of a repository comprising 1509 commits. With respect to its closest competitor, CLAIM shines most in terms of false positive reduction (-40%).
Conclusions: While not able to reconstruct a microservice architecture in its entirety, Claim is an effective and efficient option to swiftly identify microservices in Dockerized environments, and seems especially fitted for software evolution mining studies.

CCS CONCEPTS

• **General and reference** → **Metrics**; **Experimentation**.

KEYWORDS

Microservices, Static Analysis, Repository Mining, Docker

ACM Reference Format:
Kevin Maggi, Roberto Verdecchia, Leonardo Scommegna, and Enrico Vicario. 2024. Claim: a Lightweight Approach to Identify Microservices in Dockerized Environments. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024), June 18–21, 2024, Salerno, Italy.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3661167.3661206

1 INTRODUCTION

Over the past decade, microservices have gained significant popularity in software engineering academic literature [4, 12, 14]. This surge in interest seems to stem from the numerous benefits offered by microservices, such as their scalability, flexibility, and independence throughout the development lifecycle. However, alongside these advantages, microservices also present challenges, such as the additional effort required for integration and system testing.

In the context of empirical software engineering research, inquiries often adopted repository mining and source code analysis to quantitatively study different aspects of microservice architectures (MSA), *e.g.,* their evolution goal [2], microservice coupling [5], or quality evolution [13].

Despite the increasing demand for lightweight and scalable methods to identify microservices for large-scale repository mining studies, it seems as if current mining studies have to rely on either manual analyses [2] or unvalidated ad-hoc heuristics [3].

In order to fill this gap, in this work we present Claim, a static approach designed to identify microservices based on the analysis of Docker configurations files. In brief, Claim identification method relies on redacting a list of Docker services and matching them to the files originating the images. Claim then applies a set of *a priori* defined heuristics to determine which services are actual microservices, and which are instead other architectural components (*e.g.,* databases or utility services).

From the empirical experimentation conducted as part of this study, which comprises 20 repositories for a total of 13k commits and 160 microservices, Claim results to be able to identify microservices with an accuracy of 82.0%, requiring a median of 61ms to analyze a commit, and in the worst case scenario a total of 125.5s to analyze the history of a repository with 1509 commits.

The main contributions of this paper are:

- A detailed description of the approach;
- A ready to be used implementation of the approach;
- An empirical evaluation of the approach;
- A comprehensive replication package[1] of the experiments, including the entirety of the source artefacts used, accompanied by the intermediate data and final results.

2 RELATED WORK

By considering the literature on microservice-based architecture recovery, it seems as if no tool was yet developed with the specific goal of effectively and efficiently identifying microservices. In fact, approaches mostly aim at recovering microservice architectures in their entirety, by considering not only microservice identification but also the reconstruction of the dependencies between them and their relation with other infrastructural elements, *e.g.,* databases and gateways. Due to the encompassing goal considered, MSA recovery methods proposed in the literature need to rely on dynamic

[1]https://doi.org/10.6084/m9.figshare.25379212

EASE 2024, June 18–21, 2024, Salerno, Italy
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1701-7/24/06
https://doi.org/10.1145/3661167.3661206

analyses or, in some cases, a combination of static and dynamic ones. For this reason, while identifying microservices *via* such approaches is feasible, the effort required to run such approaches results suboptimal to identify microservices for mining studies, due to the recurrent need of an *ad-hoc* configuration *per* repository, and/or considerable execution times required *per* commit.

The method first utilized by Baresi *et al.* [3] to identify microservices appears to be the only one in the current literature that relies exclusively on static analysis. This approach constitutes the inspiration of this work, with Claim being a refinement and extension of it. In particular, Claim is based on the same intuition of the approach used by Baresi *et al.* [3], namely the identification and parsing of Docker *compose* files to identify microservices. The key difference of Claim lies in the strategy used to identify the original source of data, namely *compose* files (Step 1, Section 3), and the additional use of *Dockerfiles* in order to improve the microservice identification accuracy (Step 3, Phase 1-3, Section 3). Given the common goal of efficient microservice identification shared by Claim and the approach of Baresi *et al.* [3], this latter approach is used as a means of comparison for the empirical evaluation of Claim. The experimental comparison, relying on pre-defined and manually identified ground truths, also serves as a first evaluation of the efficacy and effectiveness of the two approaches.

By considering MSA recovery approaches which rely on dynamic analyses, we consider as most closely related the ones utilizing also a static process. The remainder of this section is dedicated to discussing the MSA recovery approaches of such hybrid nature, *i.e.,* the ones relying simultaneously on static and dynamic analysis.

Soldani *et al.* [8, 11] present μminer, a tool using a hybrid static-dynamic approach to reconstruct the microservice topology of MSAs. The strategy used by μminer relies on the static identification of microservices *via* parsing. However, differently from Claim, μminer relies on Kubernetes manifest files as input, rather than Docker *compose* file. This implies that it is not usable for the analysis of applications that do not use Kubernetes, a characteristic quite common in smaller or less structured projects which are often targeted by mining studies. In a subsequent dynamic phase, μminer executes the MSA to collect interactions among components by sniffing the network to generate the final microservice topology. As, differently from μminer, Claim relies exclusively on static analysis, our approach might be a more fitted option for mining studies considering thousands of commits and repositories, *i.e.,* inquiries requiring efficiency in their microservice mining processes.

Alshuqayran *et al.* [1] identify the core architectural elements to construct a MSA metamodel *via* a mix of static dynamic analyses. In a preliminary static step, similar to Claim, the approach analyses *compose* files and *Dockerfiles* to identify microservices. In contrast to Claim however, this step also requires, among others, the analysis of *Maven* files, YAML configurations, and Java source code. Apart from the language independence of Claim and more lightweight input requirements, the static portions of Alshuqayran *et al.* [1]'s approach does not appear to mention implementing strategies to identify *compose* files, nor complementing *compose* file information, a strategy characteristic of Claim (see Section 3).

Similar to Alshuqayran *et al.* [1], the approach by Rademacher *et al.* [9] also relies on statically analyzing *compose* file and *Dockerfiles* by complementing the analysis through Java source code analysis. The static microservice identification technique used by Rademacher *et al.* [9] relies on the specific Java programming language semantic and related framework constructs, which are used by the approach in order to identify the microservice endpoint. In contrast, Claim is by design independent of the specific programming language used to implement the MSA under analysis.

Finally, Granchelli *et al.* [6] developed MicroART, a tool implementing a hybrid static-dynamic semi-automatic method for MSA reconstruction. In this case, similar to Claim, their static analysis step collects information from *compose* file and *Dockerfiles*. However, differently from Claim, the static analysis of MicroART focuses exclusively on gaining an overview of component interaction, which necessitates also a subsequent dynamic analysis.

3 APPROACH

In this section, we document a detailed description of the Claim approach. Claim is based on three main phases, namely the *compose* file identification (Phase 1), *compose* file parsing (Phase 2), and microservice identification (Phase 3). An overview of Claim is depicted in Figure 1.

As background information, in MSA relying on Docker, the *compose* file is used to list Docker services, which in turn constitute the architectural backbone of the MSA. *Dockerfiles* instead are the mechanism to generate the Docker service images, *i.e.,* the mechanism allowing to spin up the architectural components of the MSA.

Intuitively, during Phase 1 Claim identifies the *compose* file where the microservices of a MSA under analysis are listed. During Phase 2, the Docker services potentially corresponding to a microservice are extracted. Finally, during Phase 3, the services are categorized as either microservices or infrastructural elements (*e.g.,* servers, gateways, databases, and monitors).

3.1 Phase 1: *Compose* File Identification

Albeit standardized guidelines are provided by the Docker platform[2], the common use of *compose* files often deviates from the predicated standards. By considering the syntactic liberty the platform itself provides, multiple *compose* files can be defined at the same time, each with its different scope. For example, commonly, separate *compose* files are used to define different development environments, *e.g.,* production and staging. Additionally, the Docker semantic allows to use *ad hoc* configuration files to specify infrastructural components and/or define specific override settings. Therefore, in order to identify the *compose* file listing the microservices of a MSA, all *compose* files need to be considered, including the ones presenting non-standard names.

Claim deals with this task by first collecting all potential *compose* files of a MSA, and subsequently selecting the ones containing in the filepath exclusively keywords associated to Docker elements (*e.g.,* `docker` or `compose`), source code (*e.g.,* `src` or `services`), development phases (*e.g.,* `dev`, `test`, or `release`), and configuration keywords of more general nature (*e.g.,* `devops`, `setup`, or `iac`).

The selected compose files are then ordered based on the likelihood of them being the *compose* file that lists the microservices of the MSA. This process is conducted by assigning a weight to each keyword used to identify the *compose* files in the previous step, and

[2]https://docs.docker.com/compose/. Accessed 29th February 2024

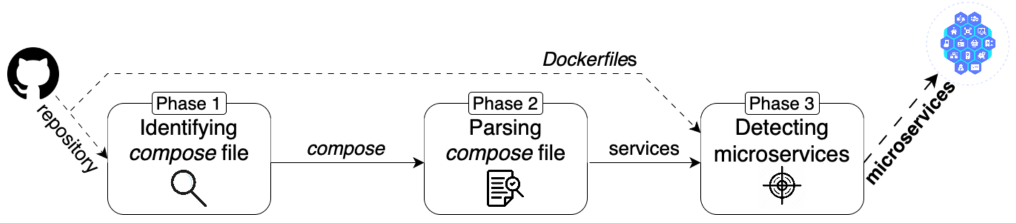

Figure 1: CLAIM workflow for the identification of microservices

subsequently sorting the filepaths in decreasing lexicographical order. Possible ties are resolved by adopting an additional ordering step on the filenames, by utilizing keywords commonly used to define *compose* files (*e.g.,* `services` or `base`). In case the selection process terminates with more *compose* files showcasing the same priority, a deterministic selection is made by considering the file with the shortest filename. *Compose* files mentioning infrastructure elements or override settings (*i.e.,* containing the keyword `infra` and `override`) are discarded, as with high likelihood they are not used to list microservices. The output of Phase 1 is the identified *compose* file listing the microservices of the MSA.

3.2 Phase 2: *Compose* File Parsing

During this phase, the *compose* file is analyzed in order to extract the information necessary to identify the microservices in Step 3.

Specifically, information regarding (i) the image run by the service, (ii) the build information of the image, and (iii) the name of the service container, are extracted from the *compose* file by following the Docker semantic reported in the official documentation.

As potential challenge of this phase, Docker may finalize the information to be collected during this phase at runtime, *i.e.,* the information may not be encoded in plain format in the *compose* file, but is rather dynamically constructed during the Docker execution.

To statically bypass this potential impediment, CLAIM uses three different strategies by leveraging the Docker *compose* semantic. First, CLAIM conducts variable interpolation to reconstruct information defined across multiple environments. Second, external *compose* files are included by recursively analyzing dependency declarations. Third, the inheritance tree of Docker services is reconstructed in order to include other possibly left implicit information.

The output of Phase 2 is the list of the Docker services of the MSA under analysis, complemented with information regarding their images, builds, and containers.

3.3 Phase 3: Microservices Detection

The final phase of CLAIM consists of identifying the microservices of the MSA under analysis. Specifically, based on the list of Docker services gathered in the previous phase, in Phase 3 CLAIM makes a distinction between the Docker services corresponding to microservices and the ones corresponding to infrastructural components.

The distinction is based on the intuition that, for a Docker service to be a microservice, the microservice source code needs to be copied from a location of the repository under analysis to the filesystem of the container. If not, this would imply that, rather than implementing a microservice, the container is utilized for a utility purpose (*e.g.,* the *Dockerfile* is used to configure a third party image).

Overall, Phase 3 is constituted by three main steps, namely (i) *Dockerfiles search*, used to identify all *Dockerfiles* of the repository, (ii) *Dockerfiles match*, necessary to map a Docker service to a *Dockerfile* if not explicit, and (iii) *Dockerfiles check*, used to assess if the source code is actually copied to the container.

3.3.1 *Phase 3 - Step 1: Dockerfiles Search.*

In this initial phase, the *Dockerfiles* of the MSA under analysis are identified by selecting all files containing in their name `Dockerfile`. False positives, most commonly consisting of scripts used to hard code build parameters, are discarded by considering the file extensions (*e.g.,* `.sh` and `.ps1`). Additional false positives corresponding to third-party services and demos are discarded by considering the presence of related keywords in the filepath (*e.g.,* `vendor`, `external`, `example`, and `demo`). The output of this step is a list of all *Dockerfiles* of the MSA under analysis.

3.3.2 *Phase 3 - Step 2: Dockerfiles Match.*

The second step of Phase 3 consists of matching the Docker services provided as output to Phase 2 to the *Dockerfiles* identified in Phase 3 - Step 1.

In the most trivial case, the Docker service is explicitly mapped to a *Dockerfile* in its definition, and hence it can be simply used as is. An additional process is instead required when a Docker service is not explicitly mapped to a *Dockerfile*, *i.e.,* developers use a custom workflow for building the image and deploying it on a registry from which the *compose* file then pulls it. In this latter case, based on the images and containers identified in Phase 2, CLAIM attempts to infer the microservice names. This process consists of leveraging different naming conventions commonly used for *Dockerfiles* and images to generate tentative names, and subsequently trying to match the tentative name to the list of *Dockerfiles* generated in Phase 3 - Step 1. If a match cannot be found, it means that the considered Docker service cannot be linked to a *Dockerfile*. In this situation, CLAIM categorizes the Docker service as not being a microservice. Essentially, if the *Dockerfile* responsible for generating the image does not exist, with high likelihood it is a third-party service. The output of this step is a list of Docker services mapped to their *Dockerfile*.

3.3.3 *Phase 3 - Step 3: Dockerfiles Check.*

After all Docker services are mapped to their *Dockerfile*, during the last step of CLAIM, we check if the *Dockerfile* actually copies source code into the filesystem of the container. In order to do so we assess if, *via* the instructions provided in the *Dockerfile*, source code files other than the ones used for configuration are copied into the container. Configuration files, in this case, are excluded based on their extension (*e.g.,*, `.xml`, `.yaml`, and `.config` files).

The final output of Claim is the list of microservices identified in the MSA under analysis.

4 EXPERIMENT

In order to evaluate the effectiveness and efficiency of Claim, we conduct an empirical experiment as part of this study. The experiment is designed to answer two research questions (RQs), namely:

RQ$_1$: [Effectiveness] *What is the effectiveness of Claim in terms of microservice identification?*

With RQ_1, we aim to evaluate the extent to which Claim is able to identify the microservices of a MSA under analysis. In order to answer this research question, we consider as evaluation metrics true positives (*i.e.,* microservices identified as such), false positives (*i.e.,* components other than microservices which are identified as such), false negatives (*i.e.,* microservices which are identified as other components), and true negatives (*i.e.,* non-microservice components identified as such). Derived metrics, namely precision, recall, and accuracy, are also utilized to discuss the results.

RQ$_2$: [Efficiency] *What is the efficiency of Claim in terms of execution time and memory consumption?*

With RQ_2, we aim to evaluate the efficiency of Claim, in terms of its execution time and memory consumption requirements. In order to answer this second RQ, we consider as evaluation metrics the time required to analyze a single commit of a MSA under analysis, the time required to analyze the entire commit history, and the volatile memory (RAM) required to run the approach[3].

4.1 Research Setting

We consider as experimental objects 20 open-source repositories, comprising a total of 160 microservices, 13k commits, and 1.7M SLOC. An overview of the repository demographics is presented in Table 1. The repositories are subdivided in two groups, according to how the *ground truth*, *i.e.,* the list of microservices contained in the repository, is established. As a ground truth regarding the *compose* files listing microservices of MSA repositories does not appear to be present in the literature, for both groups *compose* files are determined *via* a manual repository inspection.

4.1.1 Repositories with a priori Defined Microservice Ground Truth. Repositories in this group present official documentation of the microservices they contain (*e.g.,* in their README or wiki). All repositories are selected starting from the curated dataset of Rahman *et al.* [10], by using as inclusion criteria the presence of the microservice list in the repositories, and subsequently selecting the ones already used in related work (*e.g.,* [1] and [2]). The selection process leads to including 6 repositories with an *a priori* defined ground truth, for a total of 67 microservices, and 1.9k commits.

4.1.2 Repositories with Manually Determined Microservice Ground Truth. Repositories belonging to this group are the ones for which a ground truth is specifically determined for this evaluation through a manual process. In this case, the ground truth is determined by manually inspecting the source code of the repositories, and categorizing architectural components as either microservices or infrastructural elements. To mitigate the considerable manual effort required by this task, only commits involving changes to Docker configuration files are considered in this process.

ID	Repository	SLOC	#MS	#Comm.
	Microservice ground truth defined a priori			
S01	EdwinVW/pitstop	97k	7	397
S02	FudanSELab/train-ticket	344k	44	132
S03	ewolff/microservice	4k	3	140
S04	microservices-patterns/ftgo-application	21k	6	295
S05	spring-petclinic/spring-petclinic-microservices	15k	3	727
S06	sqshq/PiggyMetrics	20k	4	233
	Manually determined microservice ground truth			
S07	1-Platform/one-platform	417k	8	1.5k
S08	OpenCodeFoundation/eSchool	5k	5	273
S09	OpenLiberty/liberty-bikes	33k	3	591
S10	ThoreauZZ/spring-cloud-example	7k	3	279
S11	abpframework/eShopOnAbp	257k	7	1319
S12	aliyun/alibabacloud-microservice-demo	27k	9	422
S13	asc-lab/micronaut-microservices-poc	39k	9	384
S14	geoserver/geoserver-cloud	63k	7	1.3k
S15	instana/robot-shop	6k	7	377
S16	jvalue/ods	219k	4	1.4k
S17	learningOrchestra/mlToolKits	7k	9	1.2k
S18	microsoft/dotnet-podcasts	22k	5	728
S19	open-telemetry/opentelemetry-demo	53k	13	697
S20	vietnam-devs/coolstore-microservices	70k	4	680
	TOTAL	1.7M	160	13k

Table 1: Repository demographics (#MS = number of microservices, #Comm. = number of commits)

Repositories are selected from GitHub according to six inclusion criteria, namely, (I1) being implemented in one of the most used languages used for MSA, namely Java, Python, Go, C#, TypeScript, and JavaScript[4], (I2) being tagged with the topic "microservice*", (I3) possessing at least 20 stars, (I4) having at least 5 contributors, (I5) presenting a history of at least 250 commits, and (I6) utilizing Docker, as it is a prerequisite for using Claim. Academic demos, examples, templates for MSA, and repositories presenting a single microservice are discarded. The selection process ends with the inclusion of 14 repositories with a manually defined ground truth, for a total of 93 microservices and 11k commits.

4.1.3 Comparison Microservice Identification Approach. To gain further insight into Claim microservice detection capabilities, we compare its efficiency and effectiveness with what to the best of our knowledge is the only other purely static approach presented in the literature, namely the one of Baresi *et al.* [3] (see Section 2). As part of an ongoing work, we present such comparison as a preliminary result, while leaving a comparison with approaches using dynamic analyses as part of the future work (see also Section 6).

4.1.4 Experiment Execution. Regarding RQ_1, for each repository, both approaches are run to identify the *compose* file on all the commits history of the main branch and secondary branches already merged into the main, as suggested by Kovalenko *et al.* [7]. Given

[3]As Claim does not leverage heavy use of read-only memory (ROM), such metric is disregarded for RQ_2

[4]https://www.jetbrains.com/lp/devecosystem-2022/microservices. Accessed 11 March 2023

the outcome of this task can vary only if *compose* files are changed, commits are grouped in chunks where the set of *compose* files is unchanged. Results are then analyzed to see if the approaches select the right *compose* file. In negative case, the correct one is subsequently selected to measure the microservice detection capabilities, as this latter task evaluates the approaches on their ability of identifying microservices starting from the correct *compose* file. Only commits with a valid *compose* file are considered to evaluate the microservice identification, corresponding to 10,411 total commits.

Regarding RQ_2, the approaches are executed in their entirety (*i.e.,* both their *compose* file and microservices identification functionalities) to comprehensively measure their execution times and memory usage. To provide as realistic as possible measurements, execution times consider the analysis of entire repository histories, and include the time and memory required to checkout commits. Memory usage is masured *via* the Scalene memory profiler[5].

4.1.5 Hardware setup. All experiments are executed on a 2023 MacBook Pro equipped with a M3 Pro 12-Core Chip, 18GB Unified RAM, 494.38GB SSD, running macOS Sonoma 14.2.1 (23C71), connected to the internet with a 194Mbps download speed.

4.1.6 Replication Package. A replication package with all scripts used to define the datasets, implement the approaches, and execute the experiment, jointly with all raw data and results, is made available for scrutiny and replication purposes (see Section 1).

4.2 Results and Discussion

In this section, we present the results of our empirical experiment, accompanied by a discussion on their potential interpretation.

4.2.1 Results RQ_1 [Effectiveness]: Compose File and Microservice Identification. Concerning the ability of identifying the correct *compose* file, *i.e.,* the one listing the microservices of the MSA under analysis, Claim results to be notably accurate, showcasing a success rate of approximately 99.2%. The competitor approach [3] presents a slightly lower success rate of 94.8%. From *post hoc* manual scrutiny, we identify this lower accuracy to be mainly due to one project (S15), for which the approach of Baresi *et al.* [3] never correctly identifies the *compose* file.

Regarding Claim's microservice identification effectiveness, an overview of the results collected from the empirical experiment are reported in the form of a confusion matrix in Figure 2. From the figure we can observe that Claim is relatively accurate, reporting an overall accuracy of 82%. Claim's precision results to be lower (73.8%) due to the relatively high number of false positives (15.4%), which from manual inspection result to be mostly due to custom implementations of infrastructural elements (*e.g.,* gateways). In terms of recall, *i.e.,* how well Claim identifies all relevant positive instances, the approach behaves notably well (94.5%).

The approach of Baresi *et al.* [3] in comparison results to be less accurate (71.0%), reporting also a considerably lower precision (61.2%), while showcasing a marginally better recall value (95.0%). The main gains of Claim results to be due to its ability to detect a much lower rate of false positives (-40%) thanks to the filtering process used to categorize a Docker service as a microservice.

[5]https://github.com/plasma-umass/scalene. Accessed 29th February 2024

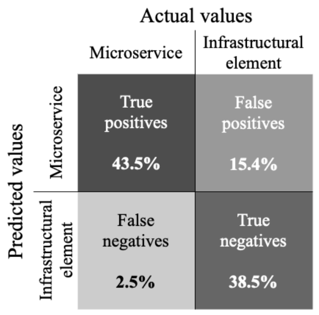

Figure 2: Claim microservice identification confusion matrix

RQ$_1$ [Effectiveness] Claim correctly identifies the *compose* file listing microservices in the vast majority of cases (99.2%). The overall accuracy of Claim in identifying microservices and infrastructural elements is 82%. Infrastructural elements are at times erroneously identified (15.4%), while microservices are not identified as such at a much lower rate (2.4%). With respect to its closest competitor [3], CLAIM shines most in terms of false positive reduction (-40%).

4.2.2 Results RQ_2 [Efficiency]: Execution Time and Memory Usage. Regarding execution time, from the empirical experiment we observe that Claim requires a median execution time of 61ms to analyze a commit, while requiring in the worst case scenario 266ms (S13) and in the best case scenario 23ms (S09).

By considering the total time required to analyze the complete history of a repository, Claim takes on median 30.7s to run the analysis (S12 and S17), while requiring in the worst case scenario 125.5s (S07) and in the best case scenario 3.8s (S03). The distribution of total execution times, compared with the ones of the approach of Baresi *et al.* [3] is documented in Figure 3. From a *post hoc* analysis, the most time-intensive task results to be the file search involved in Phase 1 and Phase 3 (see Section 3.1-3.3), which results to be especially tolling to analyze repositories of larger size, *e.g.,* S07.

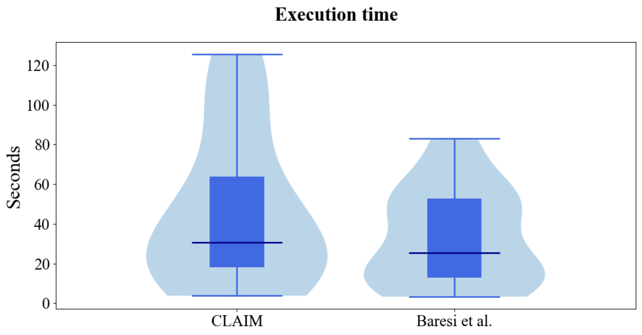

Figure 3: Total time execution

By comparing the efficiency results with the ones measured for the approach of Baresi *et al.* [3], we note that this latter approach is faster, displaying a median execution time of 38ms (S12) to analyze a commit and 25.4s (S17) to analyze the entire history of a repository.

Overall, albeit the Baresi *et al.* [3] outperforms CLAIM in the worst case scenario, given the comparable median execution time values, the overall limited time required to analyze a repository, and the considerable effectiveness gains of CLAIM (see Section 4.2.1), we deem CLAIM as a valuable option for mining studies involving repositories in the order of the hundreds.

Concerning memory consumption, CLAIM requires a limited amount of RAM, namely 20MB on median. From *post hoc* scrutiny, the most memory-intense task results to be the *compose* file loading of Phase 2 (see Section 3.2), which is more tolling for repositories with large *compose* files that load a high number of components, *e.g.,* in the case of S19. Overall, by considering modern hardware specifications, we deem the memory requirements of CLAIM minimal, and hence fitted for the vast majority of hardware setups.

By comparing memory consumption with Baresi *et al.* [3], we note that this latter approach requires a slightly larger amount of RAM (30MB on median), while displaying a more stable consumption, which fluctuates more for CLAIM according to the number of Docker services in the MSA under analysis.

> **RQ$_2$ [Efficiency]** CLAIM requires on median 61ms to analyze a commit, and takes in the worst case scenario 125.5s to analyze the commit history of a repository comprising 1.5k commits. RAM required by CLAIM is minimal, fluctuating around 20MB throughout its entire execution.

4.3 Threats to Validity

The results of the empirical experiment need to be interpreted in light of potential validity threats, as further discussed below.

Internal Validity. The results may be influenced by the experimental subjects selection. As mitigation strategy, we consider repositories obtained from a curated dataset [10], or select them *via a priori* defined criteria. Potential internal validity threats bound to the subjectivity of the manual processes involved for the ground truth establishment are mitigated by involving two researchers, with an additional researcher revising the procedure outcomes.

External Validity. Albeit measures are taken to ensure the heterogeneity and representativeness of the experimental objects, further experimentation is needed to strengthen the external validity of the results. As part of an ongoing research, additional experimentation are needed to understand how the static approach compares to the ones incorporating dynamic analyses (see Section 2).

Reliability. To ensure results can be independently reproduced and verified, all experimental material is available (see Section 1).

5 LIMITATIONS

CLAIM is designed to detect microservices exclusively in Dockerized environments. Careful considerations need therefore to be taken at research design time if CLAIM is to be used, as it may pose a considerable threat to validity, especially the external one.

CLAIM faces inherent challenges due to its static analysis nature. Unlike dynamic methods, which can benefit from additional data (*e.g.,* inter-component communication), CLAIM relies on Docker service names and *Dockerfile* paths. Consequently, architectural details are inferred using solely empirical rules based on common conventions. In an adversarial scenario, CLAIM can be led to include false positives and negatives by not adhering to common Docker configuration practices (*e.g.,* by utilizing a completely custom *compose* file naming convention). From the experiment conducted, this however does not appear to empirically be a sizable limitation.

Finally, microservice detection *via* CLAIM may experience delays if developers postpone adding microservice entries to the *compose* file until finalizing a working version. Similarly, temporarily removing microservices in the *compose* file may impact accuracy.

6 CONCLUSION AND FUTURE WORK

In this research, we present CLAIM, a lightweight microservice identification approach primarily designed for repository mining studies. The underlying ambition of this work is to provide an empirically evaluated scalable option for microservice identification. As the results of this study show, the microservice identification of CLAIM is not perfect. However, we hope this research contribution can constitute an inspiration and stepping stone towards developing both efficient and effective microservice identification techniques from which the repository mining community can profit.

As future work, we plan to optimize the accuracy of CLAIM while burdening as little as possible its efficiency. In addition, we plan to conduct an extensive empirical comparison by considering microservice identification approaches using also dynamic analyses.

REFERENCES

[1] N. Alshuqayran, N. Ali, and R. Evans. 2018. Towards Micro Service Architecture Recovery: An Empirical Study. In *2018 IEEE International Conference on Software Architecture (ICSA)*. 47–4709.

[2] W. Assunção, J. Krüger, S. Mosser, and S. Selaoui. 2023. How do microservices evolve? An empirical analysis of changes in open-source microservice repositories. *Journal of Systems and Software* 204 (2023), 111788.

[3] L. Baresi, G Quattrocchi, and D. A. Tamburri. 2022. Microservice Architecture Practices and Experience: a Focused Look on Docker Configuration Files.

[4] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. 2019. Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 187–195.

[5] D. A. d'Aragona, L. Pascarella, A. Janes, v. Lenarduzzi, and D. Taibi. 2023. Microservice Logical Coupling: A Preliminary Validation. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. 81–85.

[6] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle. 2017. MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-Based Systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 298–302.

[7] V. Kovalenko, F. Palomba, and A. Bacchelli. 2018. Mining file histories: Should we consider branches?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 202–213.

[8] G. Muntoni, J. Soldani, and A. Brogi. 2021. Mining the Architecture of Microservice-Based Applications from their Kubernetes Deployment. In *Advances in Service-Oriented and Cloud Computing*. Springer, 103–115.

[9] F. Rademacher, S. Sachweh, and A. Zündorf. 2020. A Modeling Method for Systematic Architecture Reconstruction of Microservice-Based Software Systems. In *Enterprise, Business-Process and Information Systems Modeling*. Springer.

[10] M.I. Rahman and Davide Taibi. 2019. A Curated Dataset of Microservices-Based Systems.

[11] J. Soldani, G. Muntoni, D. Neri, and A. Brogi. 2021. The μTosca toolchain: Mining, analyzing, and refactoring microservice-based architectures. *Software: Practice and Experience* 51, 7 (2021), 1591–1621.

[12] J. Soldani, D. A. Tamburri, and W.J. Heuvel. 2018. The Pains and Gains of Microservices: A Systematic Grey Literature Review. *Journal of Systems and Software* 146 (2018), 215–232.

[13] R. Verdecchia, K. Maggi, L. Scommegna, and E. Vicario. 2024. Tracing the Footsteps of Technical Debt in Microservices: A Preliminary Case Study. *Post-proceedings of the European Conference on Software Architecture* (2024).

[14] A. Villa, J. O. Ocharán-Hernández, J. C. Pérez-Arriaga, and X. Limón. 2022. A Systematic Mapping Study on Technical Debt in Microservices. In *2022 10th International Conference in Software Engineering Research and Innovation (CONISOFT)*. 182–191.