Comparison of Tools for Software Architecture Extraction of Asynchronous Microservice Systems

Jonas Frey

Institute of Information Security and Dependability (KASTEL)
Advisor: M.Sc. Snigdha Singh

English abstract.

1 Introduction

Asynchronous microservice software-systems are becoming increasingly popular for building scalable and resilient distributed applications. These systems are composed of small, independent services that communicate with each other, e.g. through messages or HTTP-communication. The architecture of an asynchronous microservice system plays a crucial role in its design, development, and maintenance. It determines how the services are organized, how they interact with each other and with the outside world, and how they can evolve and adapt over time.

In the context of continuous software engineering, where software is developed and deployed in a continuous and incremental manner, the architecture of an asynchronous microservice system is subject to erosion, i.e., the gradual divergence between the intended architecture and the implemented architecture over time [dB12]. This can happen due to a variety of factors, such as changes in the requirements, the environment, the technology, or the team. To manage erosion, we apply reverse engineering approaches to extract the architecture of the system.

One way to understand and document the architecture of an asynchronous microservice system is by extracting it from the codebase and other artifacts that describe the system. This is done for example via static code analysis, where artifacts are statically analyzed to detect the microservice components and the communication between them. Another way to extract the architecture is dynamic analysis, where we execute the system and observe the runtime behaviour, logs and other data to detect the architecture of the software-system.

In this paper, we will compare five reverse engineering approaches that are capable of extracting the architecture of asynchronous microservice systems. In section 2, we will

provide some foundation knowledge. In section 3, we will talk about the design and goal of this literature review and the selection of the papers. In section 4, we will present the results and compare them using five different aspects. We will discuss these results in section 5, followed by the related work in section 6. Then in section 7, we conclude our results.

2 Foundation

Our work is based on several foundations, which we will discuss in the sections below.

2.1 Microservice Architecture

In a microservice architecture, the software is partitioned into many small components ("microservices"), which operate independently of each other and communicate via messages [Dra+17]. This software architecture style allows the construction of highly reusable components which focus on a single task (e.g. applying a watermark to a video). This loose coupling allows for independent teams to work on different components or even the use of off-the-shelf components. Also, scaling the application can be achieved by simply duplicating the bottlenecked microservices [Dra+17]. A microservice is defined by its provided- and required-interfaces, which allow for the message-exchange with other components [SKK21].

2.2 Asynchronous Communication

[TODO: Einleitungs-Satz] [TODO: Bild]

2.2.1 Asynchronous RESTful Communication

Microservices that communicate asynchronously are typically realized in one of two kinds. Either using a RESTful pattern or using message-based communication.

Some systems use microservices that communicate asynchronously via asynchronous HTTP REST interfaces. There are two interaction scenarios for this kind of asynchronous communication. One possibility is that the initial HTTP request returns an HTTP code 202 (Accepted) and provides a location where the microservice can query the status of the operation. Once the operation on the server has finished, the provided location will return the results of the query. [MW18]

Alternatively, the microservice may be required to provide a callback method (e.g. a web hook, [Lin]) where the server can send the results once the operation has finished [MW18].

We will refer to both of these communication methods as RESTful asynchronous communication in the following paper.

2.2.2 Message-based Communication

Contrary to RESTful asynchronous communication, other microservice systems use message-based asynchronous communication. These systems use messages or events to facilitate communication between its microservices. These messages or events are typically sent using a messaging system (e.g., Java Message Service, JMS [1]) or a message broker (e.g., Apache Kafka [2] or RabbitMQ [3]). These messaging systems or brokers provide a reliable, scalable and decoupled way for the microservices to communicate with each other. Using a message broker also allows for better system performance [SKK21].

2.3 Reverse Engineering

Reverse engineering is a Software Architecture Extraction (SAR) technique that uses the artifacts of the system (e.g., source code, logs) to analyze the system. Vital for building the architecture of the software-system is information about the communication between the different components. This information is extracted using either a static (using only static inputs, e.g. source code), dynamic (using runtime information, e.g. logs) or hybrid (using both) approach. In the context of asynchronous communication, a static extraction algorithm would in the case of a RESTful asynchronous communication, analyze the HTTP calls made in code to determine the relationships between the components. In the case of message-based communication, a static approach is unable to extract a useful architecture, since message-based systems exchange those messages only at runtime and thus the required information about which components communicate with each other can only be retrieved as part of a dynamic or hybrid analysis. [SKK21; MW18]

2.4 Palladio Component Model

The Palladio Component Model (PCM) is a meta-model for the description of component-based software [BKR09]. It is used to predict the performance properties of component-based software at design-time by specifying a model of the system, its components and how the system is going to be used. A PCM consists of four main views that represent the aspects of the system. The Component View represents the functional structure of the system, i.e. the components, their interfaces and, the dependencies between them. The Resource View represents the resources (e.g., CPU, memory, databases, libraries) required by the system. The Allocation View shows, which components are allocated to which resources. Finally, the Repository View represents the data structures and operations provided by the system's components as well as the relationships between them. [BKR09]

3 Study Design

This chapter will explain the design of the systematic literature review and how it was executed.

[1] https://www.oracle.com/technical-resources/articles/java/intro-java-message-service.html
[2] https://github.com/berndruecker/Dowing-retail/tree/master/kafka/java
[3] https://www.rabbitmq.com

Inclusion	Papers that present an approach for extracting microservice architectures Papers that are able to extract asynchronous architectures
Exclusion	Papers that only present a foundation or compare other approaches

Table 1: Inclusion and exclusion criteria for selecting the papers

3.1 Study Aim

[TODO: Allgemein: was ist SLR?] The aim of this paper is to find and compare the tools available for the extraction of the architecture of asynchronous microservice systems. For this purpose, we define two guiding questions.

Q1. What are the tools available for the extraction of asynchronous architectures of microservice systems?

Q2. To what extend do the tools support software architecture extraction?

3.2 Selecting the Papers

To search for research papers, I (***[TODO: "we" or "I" here?]*** performed several queries using Google Scholar. The following queries were used to search for papers:

- `architecture (extraction OR reconstruction) (dynamic OR logs OR asynchronous) microservice`
- `("architecture extraction" OR "architecture reconstruction") (dynamic OR logs OR asynchronous) microservice`
- `reverse engineering (dynamic OR logs OR asynchronous) (microservice OR mixed-technology)`

Additionally, the references of the found results were used to look for further papers. For a paper to be selected, it had to match our selection criteria depicted in Table 1

In total, we will look at five papers, which each present an approach for the extraction of asynchronous architecture of microservice systems.

The papers are

1. ARCHI4MOM [SWK22], [SKK21]
2. MiSAR [AAE18]
3. Approach by Brosig et al. [BHK11]
4. MICROLYZE [Kle+18]
5. Approach by Mayer and Weinreich [MW18]

4 Results

Table 2 shows the results of the comparison in tabular format. In the following sections, we will discuss each paper individually.

Each approach will be presented using five aspects:

1. **Input** (e.g. source code or logs)
   The input of an extraction approach are the files or information, the approach requires to work. This can for example be static information, like the source code of the software-system, Docker configuration files and documentation, or the approach can require dynamic data, like runtime logs, communication tracing information or a list of registered services from a service discovery service.

2. **Approach** (how the extraction process works)
   An extraction approach can be broadly grouped into one of three categories. Static approaches only use static data (e.g., source code, configuration files) and do not require the system to be executed to extract the architecture. Dynamic approaches on the other hand, only use runtime data (e.g., logs, service discovery service responses) to analyze the software-system's architecture. In between, there are hybrid approaches that use both static and dynamic data to analyze the system. These hybrid approaches could for example detect the registered components statically from the source code and then monitor the runtime communication data to recognize the relationships between the components.

3. **Output** (e.g. PCM or UML) The output of the approach is whatever is the final result. For example, an approach could simply output an adjacency matrix, showing which components communicate with which other components. A more complex output could be an UML diagram, depicting the components and their relationships graphically or even a web interface showing aggregated data about the analyzed system. Another possible output is a performance model or a database.

4. **End user** (who the result is intended for) The end user is the type of person that the output of the approach is intended for. In the case of an UML diagram, a normal software practitioner could use the output to understand the system. With a performance model on the other hand, a software architect can predict the software-quality at design time.

5. **Evaluation** (how were the results evaluated) Lastly, we will look at how the different approaches were evaluated. Examples for evaluation approaches include Precision/Recall/F1 calculation or the manual comparison of the extracted architecture with the documentation. We will only focus on the type of evaluation and how it was performed and not on the results of the evaluation.

4.1 ARCHI4MOM

Input. The ARCHI4MOM approach extends the Performance Model Extraction (PMX) approach [Wal+17; SWK22] and therefore takes the same inputs. The first step in the

ARCHI4MOM approach is to instrument the source code with the Jaeger tracing tool [4] to collect tracing data. Using the OpenTracing API, ARCHI4MOM then instruments all microservices to generate trace data, which can be used later to reconstruct the asynchronous communication between the microservices. [SWK22]

Approach. ARCHI4MOM extends the Performance Model Extraction (PMX) approach [Wal+17; SWK22] to support asynchronous communication. This is achieved by adding a dependency to the OpenTracing API to each microservice to introduce a new set of information that was not present earlier [SWK22]. This tracing data is then collected in the form of JavaScript Object Notation (JSON) files, which become the input of the next phase [SWK22].

The JSON files will then be used to analyze the structure of the traces. For message-based asynchronous communication, this presents a challenge since the information is distributed over different parts of the tracing data (called "spans") whereas using synchronous communication, it would be in a single span [SWK22]. To match this inter-component communication using middleware, the called method needs to be matched using tags provided by the OpenTracing API [SWK22]. In the next step, the ARCHI4MOM approach looks for send operations that do not have information about the topic they send to [SWK22]. A topic in this case is something a receiver can subscribe to [SWK22]. After subscribing to a topic, the subscriber then receives all messages that are sent to that topic [SWK22]. ARCHI4MOM then fills in the missing topic information by retrieving it from other send operations in the tracing data [SWK22]. Then the approach iterates over all sending spans that have a "FOLLOWS-FROM" or "*message-bus*" relation tag and propagate their topics to the receiving spans [SWK22].

To reconstruct message-based communication using PMX, the authors extended PMX to support the required Palladio Component Model elements [5]. The authors then implement additional PMX logic to be able to reconstruct asynchronous architectures using these model elements. [SWK22].

Output. The output of the ARCHI4MOM approach is a Palladio Component Model (PCM) [SWK22]. This PCM contains the extracted components, as well as the interfaces for communication between each other [SWK22]. The communication channels are represented by *DataChannel*s (representing the middleware) and *DataInterface*s (representing the type of data the interface can send/receive), which are created in the PCM repository as part of the extraction [SWK22].

End User. The output of the ARCHI4MOM approach is a PCM. This model can then be used by software architects together with usage scenarios to create simulations, predicting the non-functional properties of the software.

Evaluation. To evaluate the approach, the authors created a manual PCM of the Flowing

[4] https://www.jaegertracing.io/

[5] https://github.com/PalladioSimulator/Palladio-Addons-Indirections/tree/master/bundles/org.palladiosimulator.indirections/model

Retail sample application [6]. This manual model was then verified by three developers to be correct and compared to the automatically extracted model using a Goal Question Metric (GQM) plan [Van+02; SWK22]. Using this plan, the authors then compare both sets of model elements (manual and automatic) using Precision, Recall and F1 score [SWK22].

4.2 MiSAR

Input. MiSAR extracts and gathers different data from static artifacts. This data includes docker files that assemble the containers for the microservices, docker compose files that orchestrate multi-docker-container systems, java source code, maven pom.xml files, YAML configuration files, documentation and tool support [AAE18]. The java source code is reverse engineered using a tool called Enterprise architect [7], providing UML class diagrams from the source code [AAE18]. Additionally, Zipkin [8] is used to trace communication between microservices to build a call graph [AAE18]. Information about latencies was retrieved using TCPDump [9] and information about the ports, IP addresses of container, and connectivity between containers were extracted using the Sysdig tool [10] [AAE18]. This information is all stored in a repository for further use.

Approach. MiSAR is a manual approach that is executed in two phases [AAE18]. The first phase (Recovery Design, RD) defines architectural concepts, which are extracted in the second phase (Recovery Execution, RE) [AAE18].

Phase 1 is executed in six steps. The first step is responsible for collecting the required input data, including the source code, configuration files, descriptive files etc. [AAE18]. After all data has been collected, it is stored in a repository [AAE18].

In step 2, the collected data is analyzed and combined [AAE18]. The authors extract a static view of the system in form of UML class diagrams by using the reverse engineering tool Enterprise architect on the source code [AAE18]. Additionally, the system is observed during runtime to perform a dynamic analysis [AAE18]. For this purpose, Zipkin is used to trace the communication between the different components [AAE18]. TCPDump and the Sysdig tool are used to measure the latencies of the communication and performance diagnostics at container level, respectively [AAE18].

The extracted information, together with the gathered information from step 1 is then used in the third step to determine the architectural concepts. This step focuses on bottom-up (from code to model) and top-down (from model to code) analysis to identify the architectural concepts used [AAE18].

The next step, step 4, then defines the architectural concerns [AAE18]. A concern is an area of interest with respect to a software design [BF14]. In the case of microservices, they represent common characteristics that can be implemented across multiple services [AAE18]. As these concerns are different to identify from just the code, the authors try to identify technologies that are commonly used to implement concerns that the approach

[6] https://github.com/berndruecker/flowing-retail/tree/master/kafka/java
[7] http://www.sparxsystems.com.au/products/ea/
[8] https://zipkin.io
[9] https://www.tcpdump.org
[10] https://github.com/draios/sysdig

already knows [AAE18]. For this purpose, the authors have selected the most commonly used from literature [AAE18].

The identified concerns are then grouped together in step 5, based on high level concerns that the authors have identified [AAE18].

In the last step of phase 1, the authors look at the files that were extracted for each concept and analyzed them [AAE18]. Using these files, the authors defined mapping rules that map the architectural concepts to the implementation artifacts [AAE18]. These mapping rules were then classified and grouped based on the architectural element, they output [AAE18].

Finally, in phase 2, the identified architectural concepts are extracted using the mapping rules. The results are validated and the mapping rules are refined manually, leading to an iterative process of executing phase 2 with the improved mapping rules that is repeated until the output is sufficient [Als20].

Output. The output of phase 2 is an instance diagram equivalent to an UML object diagram [Als20]. The diagram conforms to the Platform Independent Model (PIM) metamodel, which is a specific set of rules and guidelines for creating models that can be used to describe systems in a platform-agnostic way [Ben+07].

End User. The end user for the output of this approach are the software architects that can use the architecture model to understand and refine the system.

Evaluation. The approach was evaluated by comparing the output of the approach to the documentation of the actual system architecture [Als20]. The system used for evaluation was TrainTicket [ZPX+18], a benchmarking system that already provided extensive documentation about the system architecture. The authors measure the deviations from the documentation, additional elements and missing elements and calculated the Precision, Recall and F1-Score to quantify their approach [Als20].

4.3 Approach by Brosig et al.

Input. The approach by Brosig et al. uses monitoring data collected at runtime to extract the effective architecture [IWF07] of the system. For the detection of the component boundaries, either static code analysis on the source code can be used, or the boundaries can be manually defined by a software architect [BHK11].

Approach. The approach presented by Brosig et al. only extracts the effective architecture [IWF07] of the system, meaning that only parts that are effectively used at runtime are considered [BHK11]. The first step is the extraction of the effective architecture. Before the extraction process can begin, component boundaries, which separate components as single entities from the point of view of the system's architect, need to be defined [BHK11]. This can be either done manually by a software architect or automatically using static code analysis [BHK11]. After the system boundaries have been determined, the running system is monitored and the communication between the components is traced [BHK11]. The resulting *event records* (representing entries or exits of components) are

grouped together into *call path event record sets* which contain event records that were triggered by the same system request [BHK11]. This data can then be used to obtain a call path [BHK11] as well as a list of external services, a component's provided-interface calls. To extract an accurate representation of the system's components and connections between them, a representative usage profile has to be chosen, as the extraction only captures the actual communication that happens during the extraction approach [BHK11]. After the components and their connection have been extracted, the component-internal performance-relevant control flow has to be modeled. This includes the internal behavior as well as the external service calls, the component makes [BHK11].

For the second step, the approach aims to extract model parameters for performance prediction [BHK11]. This is achieved by extracting branch probabilities and loop iteration numbers from the call paths [BHK11]. For branching probabilities, mean values are used, whereas loop iteration numbers are represented by a Probability Mass Function (PMF) to allow for accurate representation of cases where a loop is for example either executed twice or ten times [BHK11]. Next, the approach tries to quantify the resource demands (e.g. CPU, HDD) of the components' internal computations [BHK11]. This demand is represented by the total processing time minus the time spent waiting for the resource to become available [BHK11]. These parameters are then averaged over the observed call paths [BHK11]. The approach is also able to handle e.g. branches that depend on input parameter values [BHK11]. In this case, the dependency has to be known a-priori for the approach to quantify these dependencies [BHK11].

In the third step, the performance model is calibrated by comparing its predictions with measurements on the real system [BHK11]. The correction to be done when measuring a deviation between the prediction and the measurement is done by increasing a factor of overhead and accounting for delays produced by the middleware stack, the system runs on [BHK11].

Output. The output of the approach by Brosig et al. is a performance model [BHK11]. In their proof-of-concept implementation, they generate a Palladio Component Model (PCM) [BHK11].

End User. The end user for this approach is e.g. a software architect, which uses the performance model to predict software quality attributes.

Evaluation. The evaluation of the approach was accomplished by implementing it and applying it to a case study of a Java Enterprise Edition application [BHK11]. The application used was the SPECjEnterprise2010 benchmark [11].

4.4 MICROLYZE

Input. Microlyze uses runtime data from a service discovery service, as well as manual inputs (e.g. semantic descriptions, mappings to technical requests) to reconstruct the software architecture [Kle+18].

[11] https://www.spec.org/jEnterprise2010/

Approach. The Microlyze recovery approach is executed in six phases, which are meant to be executed continuously [Kle+18]. The first phase rebuilds the current system architecture by checking a service discovery service (e.g., Eureka [12] or Consul [13]) and updating the status of the services in the current architecture [Kle+18].

The second phase uses the IP addresses and ports retrieved from the service discovery service to establish a link between the services and the hardware used [Kle+18]. Together with the IP addresses and ports, a monitoring agent has to be installed on each hardware component to retrieve additional information about the hardware [Kle+18]. Additionally, a monitoring probe is installed on each microservice to observe the HTTP communication [Kle+18]. The probe injects tracing data in the HTTP headers and therefore helps to gather timing data [Kle+18]. The approach then collects additional infrastructure and software-specific data (e.g., endpoint name, class, method, HTTP request) and attaches this information as annotations [Kle+18]. Using this data, the approach is then able to detect the dependencies between the microservices by following identifiers in the HTTP requests during runtime [Kle+18]. This tracing is done using zipkin [14], streamed via apache kafka to Microlyze and then stored in a cassandra database [Kle+18]. Microlyze additionally classifies each service on basis of the distributed tracing data [Kle+18]. For example, if the first accessed microservice is identical in most requests, it is classified as a gateway service [Kle+18].

In the third phase, all user transactions are stored in a database, including what the user does and in which order [Kle+18]. These user transactions are then mapped to business transactions using a business process modeller [Kle+18]. The information about the user transactions is used to create an association between the business transactions and the microservices that process these transactions [Kle+18].

The fourth phase is responsible for defining semantic descriptions to each business activity, that can be performed by a user (e.g. *register*, *open shopping cart*, etc.) [Kle+18].

After the business activities have been augmented with descriptions, phase five is able to create a mapping between the business activities ("a sequence of related events that together contribute to serve a user request" [Kle+18]) and the technical requests extracted by zipkin [Kle+18]. For this purpose, the authors enhanced the business process modeller with the regular expression language in order to describe technical requests [Kle+18]. These regular expressions are stored in the database and matched on new incoming transactions to detect, if they might refer to an already modelled business activity [Kle+18].

Finally, the sixth phase polls the service discovery service continuously to receive updates about unregistered or newly registered services [Kle+18]. Services that are no longer registered are marked as such and unknown user requests that cannot be mapped to an existing business activity using the regular expressions are added to a list of unmapped URL endpoints [Kle+18]. Changes to the underlying infrastructure are detected by changes in IP addresses or ports and lead to automatic adaptions in the architecture model [Kle+18].

[12] https://github.com/Netflix/eureka
[13] https://www.consul.io
[14] https://github.com/openzipkin/zipkin

Output. The results of the architecture extraction are displayed to the user as an adjacency matrix [Kle+18].

End User. The end users of the approach are administrators and enterprise or software architects [Kle+18].

Evaluation. To evaluate the appraoch, the authors developed a prototype and applied it to the TUM LLCM platform [15] [Kle+18]. They developed a service called *Travelcompanion* which is meant to form travel groups to save on travel costs [Kle+18]. To discover the relationships between the components, the authors produced traffic using JMeter [16] [Kle+18]. After step three completed, the authors enhanced the technical transactions with a business semantic and mapped the business activities to user transactions [Kle+18].

4.5 Approach by Mayer and Weinreich

Input. The approach uses static information, provided by configuration files and static information about services (e.g., name, version, etc.) [MW18]. Additionally, runtime communication between the microservices is collected and aggregated [MW18].

Approach. The approach presented by Mayer and Weinreich works in three steps, which are modeled by three main components [MW18]. The first step—data collection—uses the *Data Collection Library* to collect and provide static and runtime data for the architecture extraction components [MW18]. This library collects and provides service-, interaction-, and infrastructure-related information using Swagger [17] to generate API descriptions and infrastructure-specific information providers (e.g., configuration files) that are configured by the user [MW18]. The second step—data aggregation—uses the *Aggregation Service* to aggregate the information collected in the first step [MW18]. Lastly, the third step—data combination—uses the *Management Service* combines the information from the different microservices and stores it in a data model [MW18].

The authors also differentiate between three different kinds, or phases, of architecture extractions [MW18]. The first phase is the static information extraction. This extraction starts after a new service is deployed [MW18]. Using swagger, a JSON representation of the service is sent to the *Management Service*, which uses this information (namely the name and version of the service) to identify, whether the deployed service is a new instance of an already existing service, or a new service altogether [MW18]. This phase of static architecture extraction finishes, by storing all static service information in the central *Management Service* information database [MW18].

The second phase concerns the extraction of infrastructure information [MW18]. This phase builds on the information collected in the first phase and creates the according service instance node in the *Management Service*'s database [MW18]. This database is stored as a graph with directed edges, connecting the newly inserted service instance node

[15] https://www.cs.cit.tum.de/bpm/krcmar/research/finished-projects/tum-llcm-tum-living-lab-connected-mobility/
[16] https://jmeter.apache.org
[17] https://swagger.io

to the necessary services and physical infrastructure information [MW18]. If the nodes representing the physical infrastructure the service was deployed on (host and region) do not already exist, the *Management Service* creates them. As with the first phase, all information is stored in the information database at the end of the extraction [MW18].

The third and last phase is responsible for extracting the runtime information. For this purpose, all outgoing and incoming requests of a microservice are logged to a local file, including their timestamp, response time, response code, the ID of the source service instance, the URL of the target service instance, and the requested method [MW18]. The *Aggregation Service* consumes these log files periodically via REST interfaces and aggregates them [MW18]. The aggregation condenses requests that have the same source and destination instance, the same method and the same response [MW18]. The aggregated requests contain the time interval, the number of requests, and the average, maximum, and minimum response time [MW18]. Lastly, these aggregated requests are then sent to the *Management Service*, which stores them in its database [MW18]. The *Management Service* can also use this information to mark services as inactive, if it receives no runtime information anymore [MW18].

Output. The output of this approach is a database containing a directed graph that represents the system's architecture, as well as aggregated runtime information (e.g., response times) about the different requests [MW18]. This information is visualized in a web interface.

End User. The end users of the output of this approach are architects, developers, and operation experts [MW18].

Evaluation. The authors conducted a combined survey and interview study ([MW17]) and used the feedback to construct a microservice dashboard that supports the different use cases identified by the survey [MW18]. The authors then built a test scenario consisting of three microservices to test the long-term data collection capabilities of their approach [MW18].

4.6 Answering the Guiding Questions

4.6.1 Question 1

What are the tools available for the extraction of asynchronous architectures of microservice systems?

In the previous sections, we presented five approaches that are able to extract the architecture of microservice systems. Each of these systems is able to handle asynchronous REST communication and in the case of [SWK22; AAE18; BHK11], the approaches also support the extraction of message-based asynchronous communication architectures as described in subsubsection 2.2.2.

4.6.2 Question 2

To what extend do the tools support software architecture extraction?

The extend to which the approaches support the extraction of asynchronous architecture differs.

The ARCHI4MOM approach ([SWK22]) extracts the repository and part of the system model of the Palladio Component Model [SWK22]. It is not able yet to extract a usage model to complete the PCM [SWK22]. Another limitation of ARCHI4MOM is that it is not able to extract the architecture of mixed-technology systems, i.e. systems that use both synchronous and asynchronous communication between their components [SWK22].

The MiSAR approach ([AAE18]) supports synchronous, as well as asynchronous message-based systems [Als20]. Due to the manual nature of the extraction approach, the MiSAR approach is very flexible and as the mapping rules are formulated in natural language [Als20], it is possible to intervene and tweak rules that do not recover the correct architecture. The approach results in an architectural model that visualizes via a diagram the architecture of the system.

The approach presented by Brosig et al. ([BHK11]) presents and end-to-end model extraction that results in a performance model [BHK11]. Their proof-of-concept implementation showed, that it is possible to extract full Palladio Component Models, including usage models [BHK11]. The approach only considers runtime data, which means parts of the system architecture which are not active during extraction will not be represented.

The MICROLYZE approach is meant to be used continuously, which means it will be invoked on every change to the code or architecture, thus resulting in an up-to-date performance model of the system. Due to the fact that MICROLYZE keeps the architecture model data in a database, the architecture does not have to be extracted anew each time, a part of the system changes. A limitation of MICROLYZE is that it requires a service discovery service (like Eureka or Consul) to fully work. Additionally, MICROLYZE only supports REST-based communication and is not able to extract the architecture of microservice systems that use message-based communication [Kle+18].

The approach of Mayer and Weinreich only supports REST-based communication as well. Like MICROLYZE, it is meant to be used continuously and also holds model data in a database to be able to persist information between invocations. The results are visualized in a web interface.

4.7 Limitations

[TODO: Talk about limitations of the approaches]
[TODO: Talk about recommendations]

5 Discussion

The presented approaches differ in the type of asynchronous communication they support. ARCHI4MOM, MiSAR and the approach presented by Brosig et al. all support message-based asynchronous communication as presented in subsubsection 2.2.2, while

Name	Input	Approach	Output	End User	Evaluation	Year	Type
ARCHI4MOM ([SWK22])	source code	Extend PMX to support asynchronous architectures	PCM	software architect	Comparison with manual architecture	2022	tool
MiSAR ([AAE18])	source code, descriptive files, runtime traces	manual extraction approach ***[TODO: extend]***				2018	manual approach
— ([BHK11])	run-time monitoring data	combine an existing call path tracing and resource demand estimation to an end-to-end model extraction	PCM		SPECjEnterprise2010 benchmark application; comparison of prediction with measurements	2011	tool
MICROLYZE ([Kle+18])	monitoring data	continuously monitor for system changes using a service discovery service; trace HTTP requests using zipkin	database with services and their relations; web application to visualize as adjacency matrix	system administrators and enterprise or software architects	approach was applied to TUM LLCM platform, Travelcompanion service; traffic was generated and result was manually checked	2018	tool
— ([MW18])	static service information, infrastructure information and runtime logs	condense static and dynamic information into single dimension to analyze the evolution over time; visualize information in dashboard	aggregated data, visualized in dashboard		use tool in testing environment; check viability of results	2018	tool

Table 2: Results

MICROLYZE and the approach presented by Mayer and Weinreich only support asynchronous communication via HTTP REST as explained in subsubsection 2.2.1.

Additionally, the approaches can be grouped by the type of input they require. MiSAR and the approach presented by Mayer and Weinreich both require static data (e.g. source code) as well as dynamic/runtime data (e.g. tracing data, logs) [AAE18; MW18]. ARCHI4MOM, MICROLYZE and the approach presented by Brosig et al. on the other hand, only use runtime data to extract the architecture. The usage of dynamic data by all approaches is not unexpected. Asynchronous communication, message-based communication in particular, often requires the system to be executed for it to show the full communication dependencies between the microservices.

In terms of automation, all approaches except MiSAR are automated [SWK22; BHK11; MW18; Kle+18]. Therefore they are suitable to be used in a continuous integration environment, where each new iteration of code can be used to update the extracted architecture. This way, there always exists an up-to-date architecture of the current system implementation. The approach by Brosig et al. specifically, features a persistent database that stores the architectural model information [BHK11] and allows for an iterative extraction of the architecture. This means that on each invocation after the first one, the existing architecture model can be used and refined, which means that the architecture does not have to be extracted anew each time.

The MiSAR approach is specifically a manual approach with mapping rules defined in natural language. It has to be executed by a human, where some steps can be semi- or fully-automated [Als20]. A big part of the workload is done in phase 1 of the MiSAR approach, which concerns the creation of these mapping rules by iteratively defining and refining them [AAE18], although this phase can be significantly shortened, if the user of the approach decides to use the mapping rules extracted and presented by Alshuqayran et al. as a starting point. Due to the kind of this manual approach, it seems unfit to be used in a continuous software engineering environment, as at least phase 2, the extraction of the architecture using the mapping rules would have to be executed on each code change, resulting in a lot of manual work.

While the ARCHI4MOM and MICROLYZE approach both rely on a service discovery service (e.g. Eureka or Consul) to be present retrieve a list of all registered services [SWK22; Kle+18], the approach by Brosig et al. retrieves this information via static code analysis or manual input from the software architect [BHK11]. This makes the Brosig et al. approach more suitable for systems under foreign control, where it may not be possible to install such a service.

From the five presented approaches, the approach by Brosig et al. is the only one that extracts a full performance model (namely, a full Palladio Component Model) including the usage data [BHK11]. This makes it very suitable for integration in a continuous environment where the generated performance model can then be used to analyze different builds with each other.

6 Related Work

This chapter presents other papers which are similar to my work. For example [DP09], which compares different SAR approaches to formulate a state-of-the-art approach or [GIM13], which compares different SAR tools. We will also talk about the fact that [Gra+17] and [Lan+16] could be extended to support asynchronous communication in the future.

7 Conclusion

In this chapter, we will recap the findings that we made and finish the paper with concluding remarks.

References

[AAE18] Nuha Alshuqayran, Nour Ali, and Roger Evans. "Towards Micro Service Architecture Recovery: An Empirical Study". In: 2018. DOI: 10.1109/ICSA.2018.00014.

[Als20] Nuha Alshuqayran. "Static Microservice Architecture Recovery Using Model-driven Engineering". PhD thesis. University of Brighton, 2020.

[Ben+07] Gorka Benguria et al. "A Platform Independent Model for Service Oriented Architectures". In: *Enterprise Interoperability*. Ed. by Guy Doumeingts et al. London: Springer London, 2007, pp. 23–32. ISBN: 978-1-84628-714-5.

[BF14] P. Bourque and R.E. Fairley. *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society, 2014. URL: www.swebok.org.

[BHK11] Fabian Brosig, Nikolaus Huber, and Samuel Kounev. "Automated extraction of architecture-level performance models of distributed component-based systems". In: IEEE, Nov. 2011, pp. 183–192. ISBN: 978-1-4577-1639-3. DOI: 10.1109/ASE.2011.6100052.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio Component Model for Model-driven Performance Prediction". In: *Journal of Systems and Software* 82 (2009), pp. 3–22. DOI: 10.1016/j.jss.2008.03.066. URL: http://dx.doi.org/10.1016/j.jss.2008.03.066.

[dB12] Lakshitha de Silva and Dharini Balasubramaniam. "Controlling software architecture erosion: A survey". In: *Journal of Systems and Software* 85.1 (2012). Dynamic Analysis and Testing of Embedded Software, pp. 132–151. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2011.07.036. URL: https://www.sciencedirect.com/science/article/pii/S0164121211002044.

[DP09] S. Ducasse and D. Pollet. "Software Architecture Reconstruction: A Process-Oriented Taxonomy". In: *IEEE Transactions on Software Engineering* 35 (4 July 2009), pp. 573–591. ISSN: 0098-5589. DOI: 10.1109/TSE.2009.19.

[Dra+17] Nicola Dragoni et al. *Microservices: Yesterday, Today, and Tomorrow*. 2017. DOI: 10.1007/978-3-319-67425-4_12.

[GIM13] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. "A comparative analysis of software architecture recovery techniques". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2013, pp. 486–496.

[Gra+17] Giona Granchelli et al. "Towards recovering the software architecture of microservice-based systems". In: 2017. DOI: 10.1109/ICSAW.2017.48.

[IWF07] Tauseef Israr, Murray Woodside, and Greg Franks. "Interaction tree algorithms to extract effective architecture and layered performance models from traces". In: *Journal of Systems and Software* 80.4 (2007), pp. 474–492.

[Kle+18] Martin Kleehaus et al. "MICROLYZE: A framework for recovering the software architecture in microservice-based environments". In: vol. 317. 2018. DOI: 10.1007/978-3-319-92901-9_14.

[Lan+16] Michael Langhammer et al. "Automated extraction of rich software models from limited system information". In: 2016. DOI: 10.1109/WICSA.2016.35.

[Lin] J Lindsay. *Web hooks to revolutionize the web (2007)*. Tech. rep. URL: https://web.archive.org/web/20180828032936/http://progrium.com/blog/2007/05/03/web-hooks-to-revolutionize-the-web/.

[MW17] Benjamin Mayer and Rainer Weinreich. "A dashboard for microservice monitoring and management". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE. 2017, pp. 66–69.

[MW18] Benjamin Mayer and Rainer Weinreich. "An Approach to Extract the Architecture of Microservice-Based Software Systems". In: 2018. DOI: 10.1109/SOSE.2018.00012.

[SKK21] Snigdha Singh, Yves Richard Kirschner, and Anne Koziolek. "Towards extraction of message-based communication in mixed-technology architectures for performance model". In: 2021. DOI: 10.1145/3447545.3451201.

[SWK22] Snigdha Singh, Dominik Werle, and Anne Koziolek. "ARCHI4MOM: Using Tracing Information to Extract the Architecture of Microservice-Based Systems from Message-Oriented Middleware". In: *European Conference on Software Architecture* (2022).

[Van+02] Rini Van Solingen et al. "Goal question metric (gqm) approach". In: *Encyclopedia of software engineering* (2002).

[Wal+17] Jürgen Walter et al. "An Expandable Extraction Framework for Architectural Performance Models". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE '17 Companion. L'Aquila, Italy: Association for Computing Machinery, 2017, pp. 165–170. ISBN: 9781450348997. DOI: 10.1145/3053600.3053634. URL: https://doi.org/10.1145/3053600.3053634.

[ZPX+18] X ZHOU, X PENG, T XIE, et al. “Benchmarking Microservice SystemsforSoftwareEngineeringResearch”. In: *40th ACM/IEEE International Conference on Software Engineering (ICSE)*. 2018.