Computing (2021) 103:2521–2551
https://doi.org/10.1007/s00607-021-01002-z

REGULAR PAPER

Detector-based component model abstraction for microservice-based systems

Evangelos Ntentos[1] · Uwe Zdun[1] · Konstantinos Plakidas[1] · Patric Genfer[1] · Sebastian Geiger[2] · Sebastian Meixner[2] · Wilhelm Hasselbring[3]

Received: 11 November 2020 / Accepted: 7 August 2021 / Published online: 28 August 2021

Abstract

One of the chief problems in software architecture is avoiding architecture model drift and erosion in all kinds of complex software systems. Microservice-based systems introduce new challenges in this context, as they often use a large variety of technologies in their latest iteration, and are changed and released very frequently. Existing solutions that can be used to reconstruct architecture models fall short in addressing these new challenges, as they cannot easily cope with continuous evolution, their accuracy is too low, and highly polyglot settings are not supported well. In this work, we report on a research study aiming to design a highly accurate architecture model abstraction approach for comprehending component architecture models of highly polyglot systems that can cope with continuous evolution. After analyzing the results of related studies, we found two possible architecture model abstraction

✉ Evangelos Ntentos
Evangelos.Ntentos@univie.ac.at

Uwe Zdun
Uwe.Zdun@univie.ac.at

Konstantinos Plakidas
Konstantinos.Plakidas@univie.ac.at

Patric Genfer
Patric.Genfer@univie.ac.at

Sebastian Geiger
Sebastian.Geiger@siemens.com

Sebastian Meixner
Sebastian.Meixner@siemens.com

Wilhelm Hasselbring
hasselbring@email.uni-kiel.de

1 Research Group Software Architecture, Faculty of Computer Science, University of Vienna, Vienna, Austria

2 Siemens Corporate Technology, Vienna, Austria

3 University of Kiel, Kiel, Germany

approaches that meet the requirements of our study: an opportunistic, and a reusable semi-automatic detector-based approach. We have conducted an empirical case study for validation and comparison of the two approaches. We conclude that both detector approaches are feasible. In our case study, the reusable approach breaks even in terms of time and effort needed for establishing reuse, if modest reuse of detectors is possible, and is producing slightly more high quality and evolution-stable solutions than the opportunistic approach.

Keywords Modeling · Detectors · Microservices · Software architecture · Architecture reconstruction

Mathematics Subject Classification 68U35 · 68-02

1 Introduction

Microservice-based architectures are a kind of service-oriented architecture that consist of independently deployable, modifiable, and scalable services, each having a single responsibility [1,2]. Microservices typically do not share their data with other services, are deployed in lightweight containers or other virtualized environments, and communicate via message-based remote APIs in a loosely coupled fashion. They feature polyglot programming and polyglot persistence, and are often combined with DevOps practices such as continuous delivery and end-to-end monitoring (see e.g. [3–5]). Microservices are one of many service-based architecture decomposition approaches (see e.g. [6–9]). Just like other architecture decomposition approaches, they do not address the classical software architecture problems of *architecture drift and erosion* [10] well. That is, during system evolution, the architecture models increasingly diverge from the actual software as changes are made in the source code which either violate the architecture model's original specifications, or are not reflected in it, for example through the introduction of new features [11].

To address this problem, architecture reconstruction approaches have been proposed to automatically or semi-automatically produce architecture models from the source code [12–14]. Unfortunately, these approaches usually involve a substantial effort to either manually maintain the reconstructed architecture model, or repeat the reconstruction after the system has evolved (see [15]), meaning that they are not suited for supporting continuous evolution of systems. In addition, automated approaches have low accuracy (see [16]), and much additional, manual effort is needed for correcting and augmenting their results. Finally, most reconstruction approaches focus on a very limited number of programming languages and technologies (see [12]), meaning they are hard to use with modern systems, such as microservice-based systems, which use typically polyglot programming, persistence and technologies, often in their latest iterations.

For these reasons, the prospects for ever developing a one-size-fits-all, generic reconstruction method that can cope well with evolving microservice systems (and similar polyglot systems) look bleak. Fortunately, there is hope in the fact that developers usually know a lot about their projects and thus a generic, fully automated

reconstruction may not be necessary. In this paper, we report on a design science research study [17] in which we aimed to design a new approach to enable the accurate creation and continuous evolution of component architecture models in microservice-based and similar polyglot settings with little extra effort. We set out to answer the following *research questions*:

- **RQ1** How to design a 100% accurate architecture model abstraction approach for comprehending component architectures of systems that are highly polyglot?
- **RQ2** How to support continuous comprehension of such systems in the context of such an architecture model abstraction approach?
- **RQ3** How high is the required time (effort) for creating and maintaining architecture model abstractions in such an approach?

Our study was performed by first defining the design science study in terms of design context, artifacts studied, stakeholders, and their requirements. We then selected a case study for research validation and investigated the case by performing a manual reconstruction of it, used later as a ground truth. We then analyzed the related studies that fulfill our requirements best.

In particular, our work is an extension of the approach taken by [15], which is presented in more detail in Sect. 3. Based on our experience with this work, we designed an opportunistic detector-based approach which is capable of fulfilling all our requirements regarding support for polyglot, continuously evolving systems. Our evaluation of this first approach showed that it could be further refined by making the detectors *reusable*, which we proceeded to accomplish. The result were two approaches which are the key contribution of this paper:

Detectors *are software components that continuously parse relevant parts of the source code and create model abstractions from the code.*

Reusable Detectors *are detectors which can be reused across different model abstraction tasks and projects.*

We realized both approaches fully (design, prototype development, validation in the case study), and quantitatively and qualitatively compared the results of the two approaches.

This paper is organized as follows. Section 2 examines related work and explains our study's contributions in the context of the state of the art. Next, in Sect. 3 we explain the background of this study and Sect. 4 explains our research study design and the two detector-based approaches in detail. In Sect. 5 we explain the case study implementation, and in Sect. 6 we report on its evaluation. In Sect. 7 we provide a brief overview of an implementation of the same approach in a different domain, as addition proof of concept. We conclude with a discussion of the treats to validity of our approach in Sect. 8, and our general summary in Sect. 9.

2 Related work

Related Works on Microservice Architectures Microservices [1,2,8] are, among many other things, a way to decompose an architecture based on services [3]. This is an area which has been studied intensively in recent years (see e.g. [6,7,9]). According

to mapping studies [18,19], the focus of microservice research is – in contrast to our study – frequently on specific system architectures or applications, often in relation to questions of deployment, monitoring, performance, APIs, scalability, and container technologies. The problems of complexity and service composition – relevant to our study – are addressed often, but the majority of these studies focuses on a variety of qualities (with many focusing on runtime aspects) [19]. Granchelli et al. [20] provide one of the few existing microservice-specific architecture reconstruction approaches. It statically analyses Docker and Docker Compose files for names and ports, and then the Docker containers and network bridges dynamically, to reconstruct the deployed microservices from the system's communication logs. While having quite a different goal than our study, this approach confirms our thesis that microservices require different approaches to architecture reconstruction than those adopted by the existing literature on the topic. As Granchelli et al. only use information from Docker files and related data, the reconstruction achieved is much more limited than the two approaches reported in our study, but in contrast to our approach it considers information on dynamic behavior as well.

Alshuqayran et al. [21] present an approach that is intended as a groundwork for architecture reconstruction of microservices. From the analysis of 8 open source projects, the approach derived a meta-model and possible mapping rules for microservices. This approach misses the detection component, which is the major focus of our approach, but additionally focuses on a broader set of concerns than just those that can be modeled in component models. Vianden et al. [22] report on a study of a microservice-based reference architecture as a starting point for enterprise measurement infrastructures. This can be seen as an alternative to a reconstruction effort, but it requires manual maintenance of the architecture in relation to the reference architecture – which could, e.g., be provided by one of the approaches reported in our paper. Rademacher et al. [23] suggest to address the polyglot nature of microservices using an aspect-oriented modeling approach. Again, this approach requires manual effort. It could be used as a modeling extension of our approach, where our approach can deliver the information needed for creating and maintaining the model.

Related works on architecture reconstruction and abstraction Architecture reconstruction focuses on automatically or semi-automatically producing architecture abstractions from the source code [12–14,24]. Many approaches focus on identifying components or similar abstractions through automatic clustering [24,25]. A variety of approaches establish different kinds of abstractions between source code and the architecture level. Some use graph-based techniques [26], while others utilize model-driven techniques [27–29], or logic-oriented programming [14]. Other approaches [30] analyze external dependencies to discover architectures and analyze a system's quality attributes. ExplorViz observes the runtime behavior of instrumented software systems and reconstructs their architecture on software landscape and software application level [31].

Unfortunately, these approaches have some major issues in practice: (1) architecture reconstruction approaches focus on identifying abstractions from code, without considering continuous software evolution. That is, once a reconstruction effort is fin-

ished and a few subsequent evolution cycles of the software system have occurred, the reconstructed architecture is once again outdated and a new reconstruction effort would be needed. This would not be a big problem, if the reconstruction approach were low-effort, fast, and largely automated. However, (2) automated reconstruction approaches generally have rather low accuracy, precision, and recall ability. For example, a comparative study of nine approaches reports average accuracy of 31% to 58% [16]. Given the tremendous effort needed to find and correct incorrectly-mapped source code elements in large-scale systems, in practice anything else than close to 100% accuracy is hard to use. In other words, today a substantial manual effort is required to reach close to 100% accuracy when starting off with the results of an automatic reconstruction. Finally, (3) most reconstruction approaches focus on a very limited number of programming languages and technologies (e.g., only considering Java code and even there ignoring special cases such as reflection, libraries that create dynamic dependencies, dependencies injected by an external technology, and so on). Systems such as today's microservice systems use polyglot programming, persistence and technologies, often in their latest iteration; that is, different programming languages are used, and in each of them various libraries, sometimes offering multiple APIs, are used to perform tasks such as client invocations, server programming, publish/subscribe interactions, database access, dependency injection, and so on. This is combined with multiple technologies for persistence, dependency management, CI/CD, containerization, end-to-end monitoring, call tracing, and so on, each coming with their specific configuration or other domain-specific languages. New such libraries and technologies emerge constantly, which are quickly adopted by microservice projects, further complicating the issue.

For this [15] proposed an approach for creating an architecture component view from the source code using a domain-specific language (DSL) for architecture abstraction. In this approach, the architect would specify known facts, such as the names of the major components, filter patterns for the relations of packages or classes to components, and so on. The filter patterns are designed to require little or no change if the source code changes. By studying various cases, it has been shown [15,32] that this approach requires relatively little effort (compared to program size) and can cope well with the evolution of systems. As this approach seems in a number of ways more promising for practical support of continuous architecture abstraction than the existing reconstruction approaches, we decided to use it as groundwork for our study. Like the other mentioned approaches, however, it too falls short in addressing the polyglot nature of microservice-style systems.

3 Background

Our study of related works identified the approach by [15] as close to our research requirements. From an architectural point of view, this approach uses program code in Java, and the abstraction DSL, as inputs, and creates models as output. In a first step, Java source code elements are mapped to an abstract syntax tree, from which then a detailed UML model of the relevant parts of the code is created. Next, this model is interpreted and transformed. In a background analysis we investigated to what

extent it was possible to extend this Java parsing-based approach to support multiple technologies and languages. We designed a similar solution based on ANTLR[1], since that is one of the few polyglot parser frameworks that supports most of the grammars for the languages used in our case study. Unfortunately, parsing with existing grammars for ANTLR frequently failed for our case study (see Sect. 4.1.2) and other test examples, as many ANTLR parsers do not support all the latest features of all languages used in our case study.

We decided not to pursue this approach further, since it would have required sustained effort to first correct and then maintain a wide variety of grammars, just to be able to parse all language features that we might encounter. We concluded that the approach would require us to maintain a polyglot parser framework or an adapter framework to polyglot parsers.

4 Case study design

This study employs the design science research method, which supports studying the design of artifacts in a specific context [17]. A design science research study is performed in a number of design and engineering cycles. Wieringa et al. [17] defines 4 possible steps in such a research cycle: *problem investigation*, *treatment design*, *treatment validation*, and *design implementation*. Evaluation of a cycle might lead to a next cycle for improving the design. The last step of the research cycle, *design implementation*, concerns the technology transfer into the real-world context, and is optional. We have not performed it in our study. For *treatment validation*, several validation methods can be applied, including various empirical methods. In our study, we have opted for an empirical evaluation based on a case study. If empirical methods are applied, Wieringa proposes a nested empirical cycle for performing the empirical study.

4.1 Study definition

Figure 1 summarizes the main steps in our research study, which started with a *problem investigation*, followed by a definition of requirements, and a background analysis of the approach by [15] (described in Sect. 3) that our study has revealed as close to our research requirements. In parallel we performed a manual reconstruction of a case study as a ground truth (described in Sect. 4.1.2). Based on the insights of those research steps, we designed and validated first the opportunistic detector-based approach and then the reusable detector-based approach. Finally, we quantitatively and qualitatively compared the results of the two approaches.

4.1.1 Problem investigation and treatment design

In an initial *problem investigation* and requirements definition phase, we have investigated the problem from a stakeholder and stakeholder goals perspective, followed

[1] https://www.antlr.org/.

by a definition of the requirements. In parallel we have defined and studied the case study; this has influenced the problem investigation and requirements definition, and vice versa.

Context The specific *context* of our study is *comprehending microservice architectures*; this context can be generalized to *comprehending the component architectures of polyglot and evolving software systems*.

Artifact As we have argued above, to design a fully automated reconstruction technique that works well in this context is likely infeasible. Using the examples provided below, we will illustrate why this is the case in more detail. Hence, we decided to study instead as an *artifact* a *semi-automatic architecture abstraction method* inspired by the work of [15].

Stakeholders The *stakeholders* of our study are *microservice developers and architects* whose systems are complex enough to make comprehension difficult; in a broader context, any *developers and architects* who are in such a situation are the relevant stakeholders.

Stakeholder goals and requirements We first investigated high-level stakeholder goals and then derived the following concrete requirements for our design:

- **R1** The approach should support stakeholders in getting an accurate understanding of the component architecture of a system, in the form of a *complete component model* at a sufficient level of detail, i.e., a model that contains all the possible components and component types, connectors and connector types as well as the related technologies.
- **R2** The approach should lead to architecture component models that are – in the absence of human error – *100% correct*. Our approach enables 100% accuracy since it involves a full and detailed manual reconstruction of the component architecture, which is the ground truth for the development of our detectors. The process ensures that the detector developers will be familiarized with the architecture, if that was not the case previously. The detectors are developed explicitly to cover *at least* the ground truth established by the manually reconstructed architecture, and are thus guaranteed to cover all architecturally relevant elements (per R1). Please note that our approach *could potentially use heuristics as detectors.* We have deliberately not chosen this option in this article's case study. If it was chosen that detection would be less than 100% correct as a downside, but the efforts (R6/R7) could substantially be reduced this way. Investigating this option further is beyond the scope of this article.
- **R3** The approach should be applicable in a microservice setting. That is, it should be possible with a reasonable time (effort) (see Requirements R6 and R7 below)

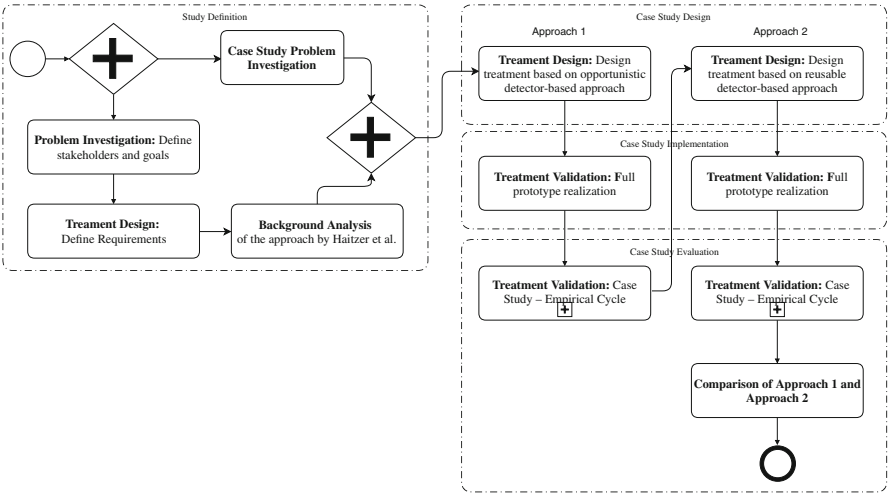

Fig. 1 Overview of the research study execution steps

to cope with *polyglot programming* and projects which frequently adopt the *latest technologies*.

- **R4** The approach should support *continuous comprehension*. That is, the time (effort) needed to recreate the architecture model after a change of the system should be minimal, i.e., usually close to zero; and in exceptional cases, a fraction of the time (effort) needed for creating the initial architecture model abstraction.
- **R5** The approach should support *traceability between architecture model abstractions and the source code*.
- **R6** Compared to the overall time (effort) needed to engineer the system, stakeholders should need to invest only a minimal amount of time (effort) for the manual part of the architecture model abstraction. We estimate that less than 1% of the development time (effort) is acceptable in practice.
- **R7** Compared to the overall time (effort) needed to manually reconstruct an architecture, stakeholders should need to invest only a small amount of time (effort) for the manual part of the architecture model abstraction. We estimate less than 10% of the reconstruction time (effort) is acceptable in practice.

4.1.2 Case study: problem investigation

In order to provide a suitable case study, based on our requirements, we require a highly polyglot microservice-based system that applies a substantial number of different technologies. The case study should have a reasonable size, but not be too large for us to be able to completely implement an architecture abstraction in the scope of a research study, maybe multiple times in each of the research cycles. It should have an industrial background (i.e., be implemented by industry experts, not a toy example by researchers). One option would have been performing an observational case study in industry. But as our study design demands that the design science artifact should

substantially evolve within each research cycle, this would have required many implementation iterations performed by industry experts to adapt the case study according to the research progress; this would have made rapid improvements of the method based on intermediate case study results impossible. For this reason, we decided to perform a so-called *mechanism experiment* [17], i.e. implement the architecture abstraction prototypically for the case ourselves.

We selected an open-source system[2] that was built as a demonstrator for the Instana monitoring technology. We report here on the master branch from 2019-10-23. Overall it consists of 140 files with a total of 5311 lines of code. It was built by industry experts from the company Instana in the timeframe Jan, 2018 – Oct, 2019; hence we believe it to be a good representative example for the current industry practices in microservice-based architectures. The project is highly polyglot: it consists of services written in JavaScript/NodeJS, Java/Spark, Python/Flask, Go, PHP/Apache, RabbitMQ messaging, and Python/Go AMQP messaging. These services use Redis, MongoDB, and MySQL as database technologies, accessed with various APIs for RESTful HTTP communication. AngularJS is used for the web frontend. Nginx is used as an API gateway and web reverse proxy. Docker, Docker Compose, Docker Swarm, and Kubernetes are used for lightweight virtualization and autoscaling. DC/OS and OpenShift are supported. End-to-end monitoring via Instana is supported, and some services have Prometheus metrics endpoints. A load generator is built with Python/Locust. Paypal is used as an external service.

For problem investigation, we performed a full manual reconstruction of the component architecture of the system as a ground truth for the case study. Figure 2 shows the result, a detailed component model specifying the component types (e.g., *Services*, *Facades*, and *Databases*), and connector types (e.g., *RESTful HTTP*, *Synchronous/Asynchronous*, *database connectors* etc.). This figure is based on the auto-generated figure created by the prototype implementing our proposed approach, described below. The system consists of 18 components and 29 connectors. More specifically, a *Client*, a *Web UI*, an *API Gateway* as entry point of the system, seven *Services*, three *Databases*, a *Message Broker*, an *External Component (Service)*, two *Monitoring Components*, and a *Tracing Component*. We kept precise time records of the manual reconstruction effort. It took us *2468 minutes (approx. 5 person-days) to perform the reconstruction*.

To study R3, for a very rough comparison of time (effort), we have used the numbers estimated by COCOMO [33] that would be needed for constructing the case study in an industry setting. We used the online calculator provided by COCOMO [3]. To be on the safe side, we used very conservative parameters for the COCOMO estimations (assuming only nominal values for parameters such as *experiences*, *capabilities*, *developed for reusability*, and so on). In total, the estimated effort for the case study system was *23.7 person-months*. This estimation is inline with the estimation in Code Complete which states: "The industry-average productivity for a software product is about 10 to 50 of lines of delivered code per person per day (including all noncoding overhead)." [34]: Assuming 1720 working hours a year, the 23.7 person-months would

[2] https://github.com/instana/robot-shop.

[3] https://csse.usc.edu/tools/COCOMOII.php.

yield 1698.5 person-hours or 212.31 8-hour person days. This means 31.43 lines of delivered code would be needed for the COCOMOII estimate, which is very close to the average of the Code Complete estimate of 30.

4.2 Detector-based architecture abstraction approaches

In this section we present and describe in detail the design for *Approach 1* and *Approach 2*. The code and models used in and produced as part of this study have been made available online for reproducibility.[4] Both approaches are based on detectors and aim to address architecture reconstruction challenges introduced by continuous evolution of microservice-based systems and their polyglot nature. Approach 1 is more case-specific and requires custom detectors, while Approach 2 provides detectors that can be reused in multiple cases.

Note that both approaches presuppose that a system expert (architect) has identified the high-level, component-and-connector architecture of the system—with which he should be familiar either way—and modelled it in an execution script that iterates the detectors over each system element. This involves a relatively small per-release effort (removal and addition of services and links between releases, cf. R4, R6, R7 and Sect. 6.2), but can also be obviated altogether by adapting the detector approach to this domain, as shown in Sect. 7.

4.2.1 Approach 1: opportunistic detector-based architecture model abstraction

The design used in the study was based on small detectors, one for each feature relevant for detecting one or more architecture abstractions. For example, if code is written in JavaScript/NodeJS, importing the `request` library means that a RESTful HTTP call could possibly be used in the file(s) using this specific technology; if a `request(...);` is present in addition, an HTTP call is actually made in the file. If a manual inspection confirms that a RESTful HTTP call is actually made, we have precise evidence for the presence of a RESTful HTTP call and can establish traceability links to all occurrences of such invocations. Based on such simple detectors, we can correctly detect most evolution scenarios: if changes to other parts of the file are made, it is not possible that the detection of the RESTful HTTP call will fail. If another RESTful HTTP call is added, it will be detected, too. If all RESTful HTTP calls are removed, the detection will fail, as it should, and manual action is required. Only if a RESTful HTTP call with a different technology is made, would a remodeling of the detector be required.

This new approach would not work better than the one from Sect. 3 in terms of parsing, if we followed the same full-fledged parser-based approach. However, some parser frameworks support scanning for the occurrence of parse rules rather than requiring parsing the whole file. One such parser framework is *pyparsing*[5], which we used to only parse the relevant parts of the code. This solved the parsing issues described in Sect. 3. To illustrate the approach, let's consider a simple detection from our case study.

[4] https://doi.org/10.5281/zenodo.5235931.

[5] https://github.com/pyparsing/pyparsing.

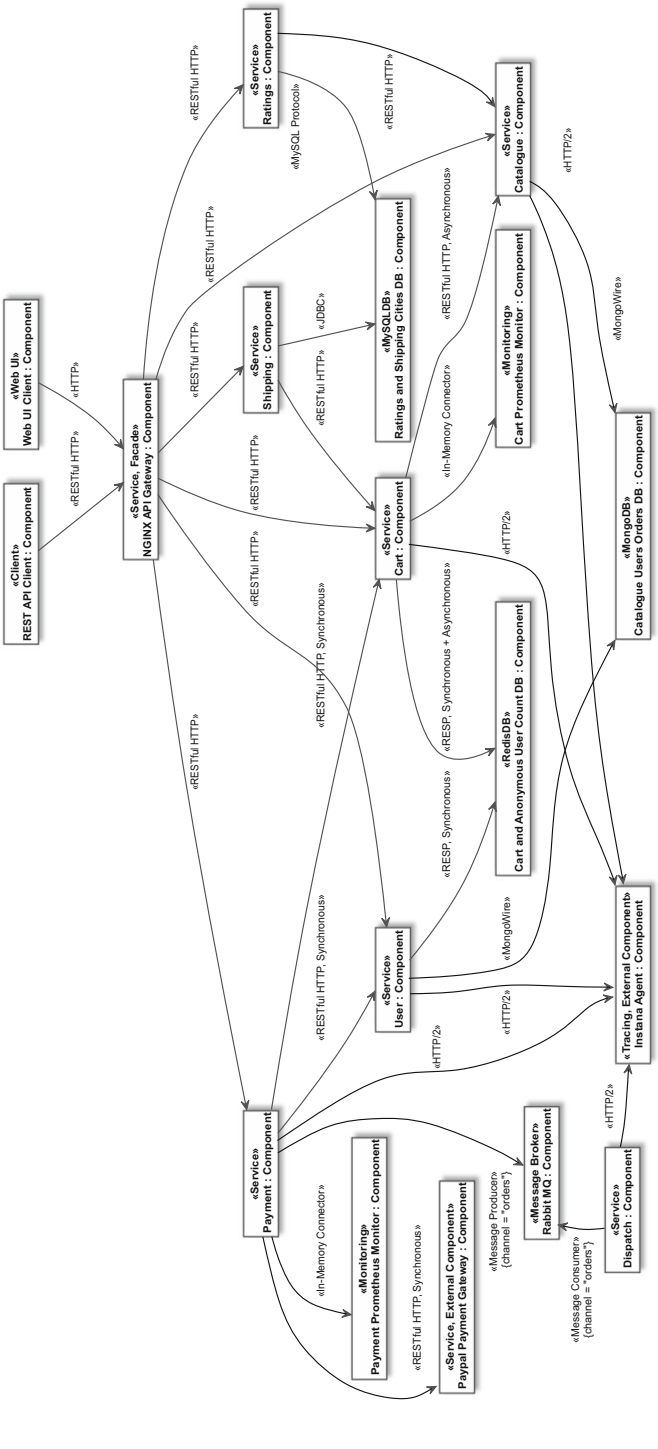

Fig. 2 Case study: overview of the reconstructed component architecture Ground Truth as a UML2 model

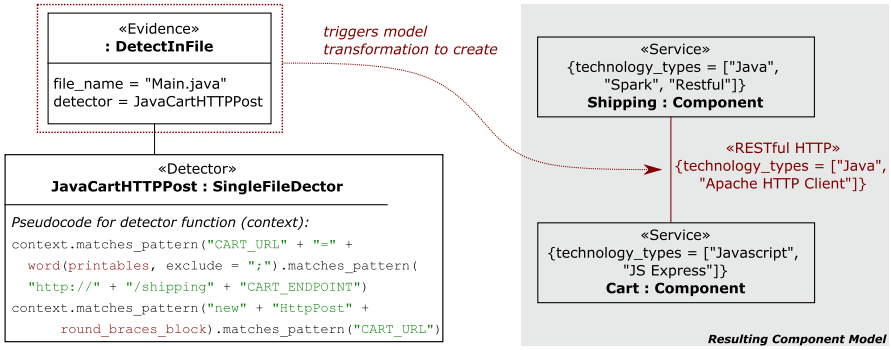

Fig. 3 Opportunistic detector example: detecting a restful HTTP connector

Assuming that we have previously detected two components, the *Shipping* and *Cart* services (cf. Fig. 2), we now want to determine the presence, and the technology, of any connector between the two components. For detecting these two components we used the *JSExpressService* and *JavaSparkService* detectors that return an evidence specifying the corresponding technology types. The detector process will call the detector instances listed in the *DetectInFile* evidence to determine the connections between the two components (see Fig. 3). In this case, if the *DetectInFile* evidence is successful in detecting (1) a *Main.java* file in the specified directories, and (2) successfully executes the *JavaCartHTTPPost* detector, it adds a connector of the *restfulHTTP* type between the *Shipping* and *Cart* components to the model. Here, the *DetectInFile()* represents the reusable code of the detector process, and *JavaCartHTTPPost* is a specific detector required for this particular occurrence (i.e., detecting the presence of a Java HTTP call from *Shipping* to *Cart*). As the specific code will be different for other occurrences, we call this the *opportunistic approach*. Please note that usually, the specific code required for a detection is rather small, e.g. in this case two lines of pseudo code are enough.

As we will discuss in more detail below, following this approach, we were able to design a solution that fulfills all the requirements of our study. However, we observed that, as can be seen in the *JavaCartHTTPPost* example, a lot of code is very specific for the particular case at hand, and that many aspects of the detection could be automated to a higher degree, with a higher code reuse. For instance, in the simple example given here, most likely many Java posts, or even other HTTP requests using the same API, could be detected with a more generic detector. If this detector is selected, it is known that it produces *Restful HTTP* connector links; thus a reusable solution could provide this knowledge as default value. Also, it might not be necessary to specify the exact file in which the request occurs, as this could be "guessed" from the directories of the detected components. As a consequence, while the approach described here works well, a reusable detector approach with less specification effort per case and more automation potential might be possible. Consequently, we aimed to design such an approach next.

That is, based on the current state of the art, this approach cannot satisfy Requirement R3 (concerning a "small amount of time (effort) for the manual part of the

architecture abstraction") and makes Requirement R4 hard to achieve ("continuous comprehension"). Based on this experience, we designed an opportunistic detector-based approach (*Approach 1*) to cover all challenges that the approach by Haitzer and Zdun cannot address. Based on this, we realized that it is also possible to design a similar, but reusable, detector-based approach (*Approach 2*).

4.2.2 Approach 2: reusable detector-based architecture model abstraction

In the reusable detector-based approach, we aimed to reduce the necessary specification in the abstraction model and completely get rid of any case-specific detector code. Instead, all detection should be handled in reusable detectors. We managed to bring almost all specifications of architecture abstractions down to a single line of code per abstraction. To illustrate this approach, let us again consider the previous example. We developed a generic, and hence reusable, detector for the Java Apache HTTP technology, which we provide to the method, along with the IDs of the two components, to create a link between them. The directory in which to search for the link is taken from the component, where it is provided as a top-level directory only (no specific directory or file names are provided anymore).

Given this dramatic reduction in the code size that needs to be written by users, it might seem at first that full automation might be possible. However, this is not correct: Note that the detector specification is meant as an *assertion by a human* that a component or connector link of a certain type was found. With this little extra information – which is much easier to obtain than performing a full manual architecture model reconstruction – we can avoid the issues that make automatic detection hard or even impossible. The problematic part that requires human input in this case is the *Cart* URL and endpoint, which are two specific variable names used in the Java implementation (see pseudo code in Fig. 3). We could potentially guess them to a certain extent, but developers could find many ways to implement or change them in the future, such as hard-coding the URL directly, obtaining them through a call to an arbitrarily named method, reading them from a file, and so on. By requiring the human identification of the occurrence once, we greatly reduce the possibility of any false positives or negatives.

In our reusable detector, we use some heuristics, and the human user who specifies the case-specific detection must be aware of the heuristics and their limitations. For the example detection between the *Shipping* and *Cart* services we used in Sect. 4.2.1, the *JavaApacheHTTPLink* reusable detector. It is able to find matches for *get*, *put*, *post*, and *delete* requests, as illustrated in Fig. 4. For all of these, it checks in the relevant Java files (`file_endings = ["Java"]`) if a respective *new* statement is found. If so, we have previously auto-detected possible aliases for the component names in various places; e.g., in the specific case, these were detected in Docker *env* statements. If one of the HTTP method matches contains a match that links to one of the target component's aliases, we have found a match for calling from source to target using an HTTP method. Otherwise we use the detector method *get_var_assignment_matches_containing_url_alias* to find all Java variable assignments that match one of the target aliases. If one of those is used in a match, it also constitutes a match for calling from source to target component. Both are seen as evi-

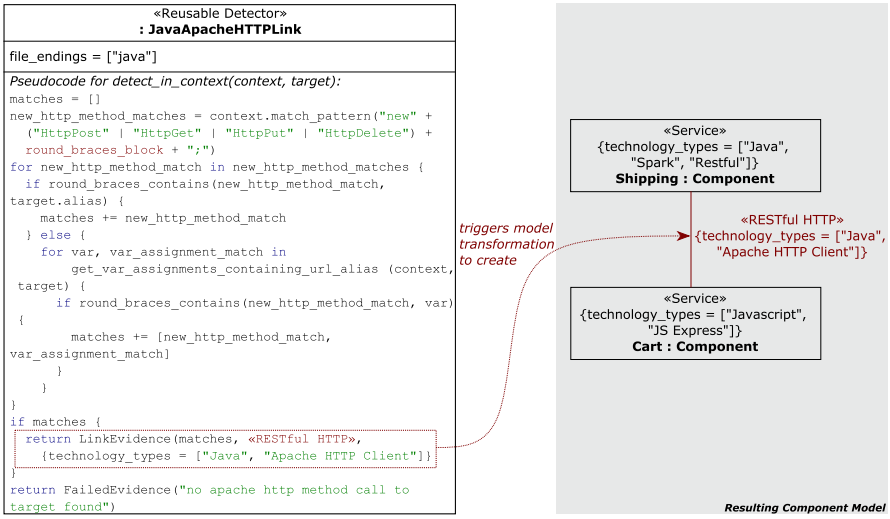

Fig. 4 Reusable detector example: detecting a restful HTTP connector

dences for a link between the source and target components. This would then trigger the model transformation for creating the Restful HTTP connector in the component model.

With this design, the detector process can tolerate many changes in system evolution (even more than in the design from Approach 1). For example, calling the URL directly instead of using the *CART_URL* variable, moving to a different variable name, moving the call to another method or file, and so on, are examples of possible non-breaking evolution scenarios. The design also fulfills all requirements set for the design study. Thus, we next studied how the two approaches compare with regard to our requirements in the context of our case study to answer the research questions.

5 Case study implementation

5.1 Architecture UML profile

As the generation target of our model transformations we introduced a UML profile. In it, components are extended by the stereotype *component types* (see Fig. 5), and connectors by the stereotype *connector types* (see Fig. 6), for introducing the microservice-specific modeling aspects. That is, to be able to apply our two approaches, we first performed an iterative study of a variety of microservice-related knowledge sources, and we refined a meta-model which contains all the required elements to allow an adequate reconstruction of architecture model abstractions in the microservice domain. The resulting stereotypes range from general notions such as the *Service* component type, to very technology-specific classes such as the *RESTful HTTP* connectors. As component types we support, for example, *Service*, *Pub/Sub*, *Message Broker*, *Event Sourcing*, *Steam Processing*, *Client*, *External Component*, *Web UI*,

Monitoring, *Tracing*, *Logging*, *Saga Orchestrator*, and various kinds of *Databases*. As connector types we support, for example, various *Service Connectors* such as *RESTful HTTP*, *SOAP*, or *GRPC* connectors, various *Web Connectors* such as *HTTP*, *HTTPS*, or *HTTP/2*, various kinds of *Synchronous* and *Asynchronous* connectors, various *Indirect* connections, *In-Memory* connectors, various *Database* connectors, various *Event-based Connectors* such as *Publishers* and *Subscribers*, and various *Messaging Connectors* such as *Message Producers* and *Message Consumers*.

5.2 Detector framework

One of the contributions of this paper is a model for the design of a semi-automatic detector framework for creating architecture abstractions along with traceability links to the source code. The latest design for this part of our study in Approach 2 differs only in details from the one from Approach 1. Thus we report only the latest design here, as it is a few refactoring cycles ahead of our earlier design. As shown in Fig. 7, a number of detections are abstracted in a *Project*. Each detection traverses the models to detect features of interest. That is, to the project we add architecture abstraction in specifications such as those in Fig. 4. In those specifications, the *Detectors* that the project should use to detect the abstractions are specified. For example, the 2 components (*Shipping* and *Cart*) in Fig. 4 use the detectors *JavaSparkService*, *JSExpressService*, and the connector between them uses the detector *JavaApacheHTTPLink*. Two specific subclasses of *Detector* are shown in Fig. 7: one for detecting at least one matching file and one for detecting matches across multiple files. The former is used as superclass for the majority of our current detectors; the latter is used occasionally. As illustrated in Fig. 4, detectors use *DetectorContexts*, which implement the scanning and matching methods and contain the text to be parsed. If detectors find matches, a *Match* is used to store the matching text, its position in the parsed text, the file, and the directory; this way, traceability to the source code is established. When all required matches are found for an architecture abstraction, the detector creates an evidence for it (like the *LinkEvidence* in used in Fig. 4). As shown in Fig. 7, *Evidence* has three main subclasses, called *FailedEvidence*, *NamedEvidence*, and *LinkEvidence*: *NamedEvidence* has an additional subclass *ComponentEvidence* in which possible link types can be collected on the component if its matches can be considered as enough items for a link (e.g., a Web server, offering the component already, implies possible links to the Web clients). It has also a subclass *ServiceEvidence* to specify the corresponding component type.

The project stores detected components to use them for later detection; e.g., the directory guessing in link detectors explained above works this way. Our design only works for components and connector links so far; for supporting other abstractions, more evidence types and additional functionality on the project would be needed. The project stores all failed evidences, because we do not stop the detection if one failure occurs, but rather let all detectors run through and provide the user with all failures that occurred.

As proof of concept, we realized a process, using Python scripts, that takes the polyglot source code and the detector specification as inputs. It creates an architectural

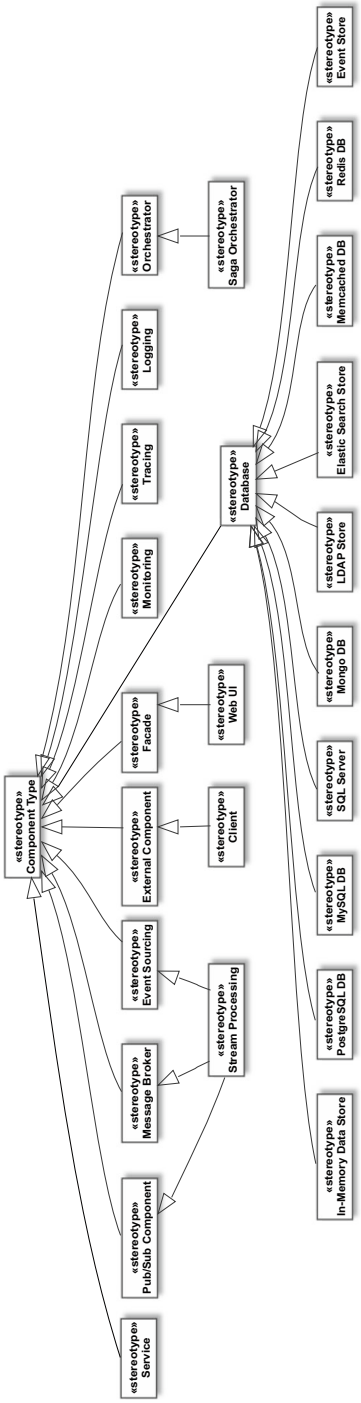

Fig. 5 Architecture UML profile: component stereotypes

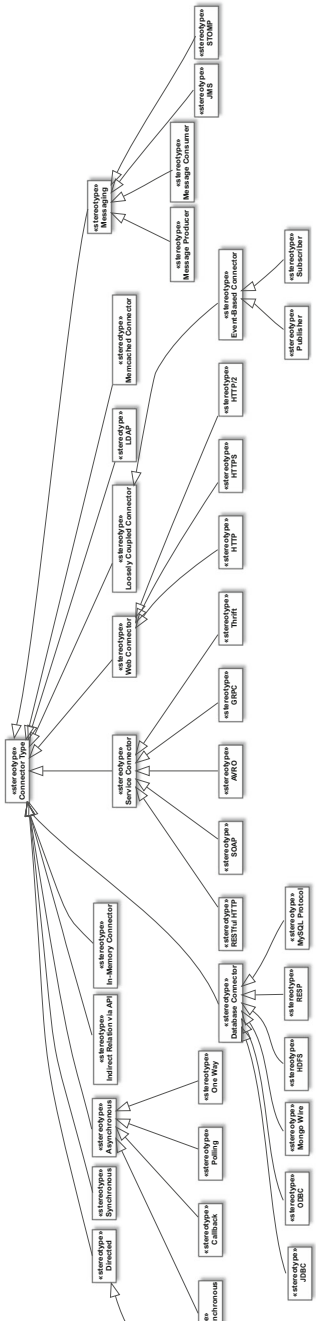

Fig. 6 Architecture UML profile: connector stereotypes

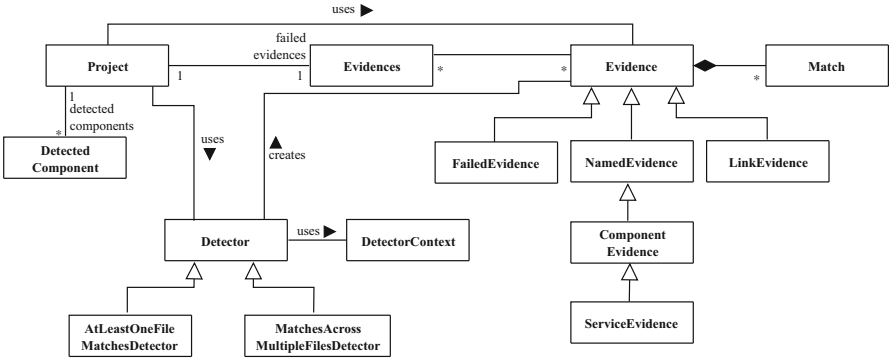

Fig. 7 Resulting design: domain model of the detectors

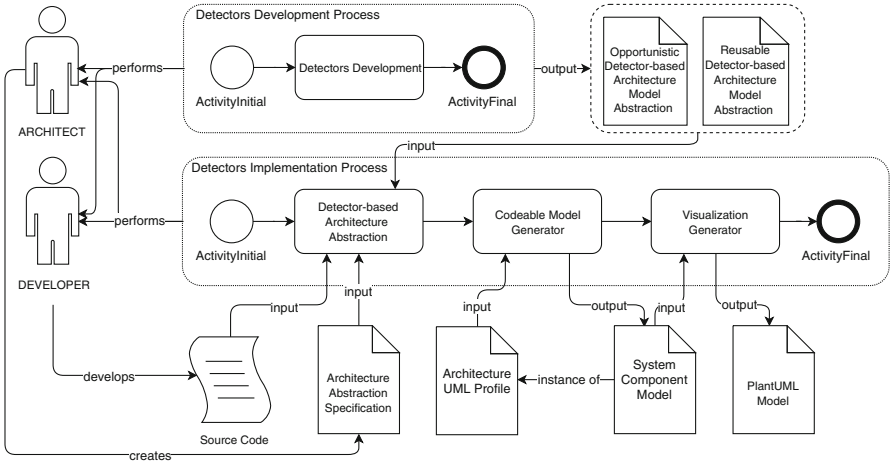

Fig. 8 Process flow architecture of the prototype

abstraction containing a UML-style component model together with *Evidences* and *Matches*. Models are expressed using the Python modeling library CodeableModels [6], a Python implementation for precisely specifying meta-models, models, and model instances in code with a lightweight interface. The process contains a visualization generator for generating PlantUML diagrams such as the one in Fig. 2.

While the process flow could be changed substantially compared to how we constructed our prototype, the general process flow architecture illustrates how the building blocks interact in order to enact the design explained above (see Fig. 7). The process aims to generate models for the modeling library CodeableModels. As illustrated in Fig. 8, the performer of the process is envisaged as a developer or architect. The architect specifies the architecture abstraction specification for a software system, while the developer implements it in source code. Both roles can also be involved in the development of detectors for the two approaches outlined above. The *Detectors*

[6] https://github.com/uzdun/CodeableModels.

Development Process produces as output either the *Architecture Model Abstraction* based on the *Opportunistic Detector* or the *Reusable Detector* approach, which in turn are used as inputs in the *Detector-based Architecture Abstraction* phase of the *Detectors Implementation Process*. The architecture specification and the system's source code are also inputs of the *Detector-based Architecture Abstraction* phase. The detector performs the architecture abstraction, and when successfully completed, uses a code generator to generate the corresponding *System Component Model* with CodeableModels. The code generator utilizes the *Architecture UML Profile* to instantiate the *System Component Model* from it. Finally, CodeableModels contains a *Visualization Generator*, which uses the *System Component Model* as input, for generating PlantUML diagrams such as the one in Fig. 2.

6 Case study evaluation

6.1 Effort and size

In this subsection, we want to give a rough estimation of the effort required to design and implement prototypes for (1) the generic detectors approaches for our two approaches and (2) the case-specific code in both approaches as well as a size comparison in terms of lines of code (LoC). We discuss in Sect. 8 why such a comparison can only provide a rough estimate. More research is needed to generate solid numbers, e.g. for precise effort prediction. However, for the purpose of this study, roughly correct numbers are good enough, as we are interested in understanding the effort and size relations in orders of magnitude. The efforts in minutes reported in Table 1 are based on manual time recordings made throughout our project. Lines of code are automatically counted using the VS Code plug-in VC Code Counter, which supports the counting of only the Python code.

As can be seen, the generic code base needed for Approach 2 is substantially larger (64.41%) than for Approach 1 and we needed moderately more time (21.62%) for creating it. The effort increase is less than the code size increase, as the generic code base contains a common code base of about 40% of the code, which was created in 1900 minutes. In addition, a learning effect from the experience in the design and implementation of Approach 1 probably also played a role in the reduction. We believe that this learning effect is small, as the non-common code parts turned out to be significantly different. In addition, Approach 2 contains reusable detectors as part of the generic code, which Approach 1 does not; it contains the case-specific detectors instead. Note that for our case study, the number of reusable detectors in Approach 2 is very high because we have studied a highly polyglot case, and started out with zero detectors; for a less polyglot case (as the cases discussed in Sect. 7), or if already extant detectors can be reused (e.g., from an open-source detector repository based on our approach), would dramatically reduce the number of new detectors and the concomitant effort expended. As a consequence of these numbers, the generic code effort is much higher (146.78%) for Approach 2, and the LoC needed for Approach 2 are significantly more (338.89%).

The situation for the case-specific parts is reversed: The case-specific model for Approach 2 is much smaller (−43.69%) than for Approach 1, and requires much less effort (−45.13%). In addition, Approach 1 requires 107 LoC for case-specific detectors constructed in 870 minutes (which are totally absent in Approach 2). Consequently, the case-specific code for Approach 2 is in total significantly smaller (−62.94%) than for Approach 1 and required substantially less effort (−79.13%). That means that realizing a reusable solution can pay off in the long run, when the approach would be applied on many projects. If the approach is needed only once (and for a small-scale case study as performed here), the reusable approach in Approach 2 does not pay off, as the total effort and LoC comparisons show.

6.2 Requirements fulfillment

For Requirements **R1** and **R2** we can assess that both approaches are able to exactly reproduce the component model from the ground truth. The same result would likely be possible with the approach by Haitzer and Zdun (discussed in Sect. 3), but this has not been fully implemented in our study. Our approach can lead to highly accurate architecture model abstraction since no additional manual effort was needed to correct the resulting model. Moreover, the developers who create the detectors are very familiar with the system, meaning that they hold in depth the system characteristics and requirements. Requirement **R3** is about supporting a highly polyglot setting, as is typical of modern microservices. This seems infeasible for the approach by Haitzer and Zdun with regard to required time and effort, unless the state of the art on polyglot parser approaches and support for multiple well working grammars is significantly improved. Both new Approaches support this requirement well.

Requirement **R4** is about continuous comprehension. In order to test this, we have examined the differences in architecture and technologies used in the system in four additional releases: one prior to our case study, three later ones. We did not only test evolution in later releases, but also in prior ones, in order to be able to test whether our approach works for the removal of features. These are summarized in Table 3. Our approaches are able to detect (a) the removal of services, (b) the removal of system-level capabilities (e.g. Prometheus), and (c) the modification of specific technologies (e.g. Go). As can be seen in the table, the additional time effort (in minutes) to the script containing the abstracted architecture model (execution script) to each release is negligible (a few minutes). However, there is a caveat for Approach 1: Due to the many small opportunistic detectors, different implementations for similar concerns and opportunistic code reuse (copy & paste) occurred. It is known that this leads to problems during evolution, such as changes not being carried through for all similar code fragments, or that overly specific code for particular detections can lead to breaking detectors.

Requirement **R5** concerns traceability from code to models. Both approaches establish trace links automatically. Requirement **R6**, i.e., how the amount of time (effort) for creating the architecture abstractions compares to the overall amount of effort for engineering the system, is fulfilled for Approach 1 and Approach 2: Compared to the COCOMO II estimate for delivering the case study project as an industrial solution,

Table 1 Effort and size comparison

	Approach 1 effort in minutes	Approach 2 effort in minutes	Diff.	Approach 1 lines of code	Approach 2 lines of code	Diff.
Generic code base	5329	6481	21.62%	576	947	64.41%
Reusable detectors	–	6670	N/A	–	1581	N/A
Total generic code	5329	13151	146.78%	576	2528	338.89%
Case-specific detectors	870	–	N/A	107	–	N/A
Case-specific Model	534	293	−45.13%	206	116	-43.69%
Total case-specific code	1404	293	−79.13%	313	116	-62.94%
Total	6733	13444	99.67%	889	2644	197.41%

Table 2 Comparison between the two approaches on requirement fulfillment

Requirements	Approach 1: case-specific detectors	Approach 2: reusable detectors
Component model reconstruction (R1&R2)	100% correctness (if detectors are not heuristics)	100% correctness (if detectors are not heuristics)
Polyglot support (R3)	Fully supported	Fully supported
Continuous comprehension (R4)	Fully supported	Fully supported
Traceability (R5)	Automatically ensured	Automatically ensured
Estimated percentage of overall system development effort (R6)	0.7%	0.1%
Comparison to manual reconstruction (R7) as percentage of manual reconstruction	56.98%	6.76%

the efforts both for Approach 1 and Approach 2 are tiny. For instance, assuming 140 work hours per month, the case study construction would have consumed, according to COCOMO II, 199080 minutes. That is, the case-specific effort for creating a model and detectors for Approach 1 would be 0.7%, and for Approach 2 0.1%. That is, according to our COCOMO II estimates both approaches are way beyond the set target of 1%. Please note that this does not work out, if generic code base and reusable detectors cannot be reused. Then we observed 4% for Approach 1 and 7% for Approach 2, which might still be acceptable for some projects, but are beyond our 1% target. For more extensive systems these numbers would be much smaller in comparison. Requirement **R7** is about the comparison of the approach to a manual reconstruction. If we compare to the manual reconstruction effort for the case study (see Sect. 4.1.2), we can see that Approach 1 requires a case-specific effort of 56.89% of the manual reconstruction effort; Approach 2 only requires 11.87% of the case-specific effort. Thus both approaches, when combined with a single manual reconstruction (or with our approach applied permanently from the inception of the project onwards), would require much less effort than periodically repeated manual reconstruction efforts. Assuming the existence of a large repository of reusable detectors (e.g., as an open-source project), Approach 2 would be vastly superior to Approach 1, too; without it, the substantial effort needed to create reusable detectors might eat up much of the benefit compared to Approach 1.

Table 2 summarizes and compares *Approach 1* and *Approach 2* based on the stated requirements. It is evident that, although both approaches meet the requirements we have set, there is a considerable difference in performance in terms of effort in favor of Approach 2.

Table 3 Continuous comprehension support (release examined in case study in **bold**)

System release	16.04.2018	**23.10.2019**	06.07.2020	26.08.2020	22.02.2021
System lines of code	3650	**5311**	5115	4997	5746
Number of system elements	12 components 17 connectors	18 components 29 connectors	18 components 29 connectors	18 components 29 connectors	18 components 29 connectors
System changes	• No Ratings service yet	• Ratings service has been added	• Dispatch service has been modified	• Shipping service has changed framework from Java Spark to Spring Boot	• An Event Listener has been introduced in Ratings service.
	• Prometheus, Paypal Payment, Instana not yet added	• Prometheus, Paypal Payment, Instana have been added			
Time needed (in minutes) to adjust execution script (Approach 1 / Approach 2)	8 min/5 min‡	**NA**	5 min/3 min†	5 min/3 min†	5 min/3 min†

†Compared to the previous release, ‡Compared to the case study release version

7 Extending the approach to cases from different domains

To further assess whether our approach is also applicable in different case study settings, we applied it to two cases in the domain of modeling inter-service communication in API-centric communication models. We did this with the purpose of automatically detecting *asynchronous cycles in communication at the API-level*. These *unintended domain-based cyclic dependencies* [35] manifest mainly on the conceptual level and less on the implementation level and are therefore considered relatively difficult to track exclusively through static code analysis methods. However, collecting runtime information to track these dependencies can often be very time-consuming. Hence an approach that focuses on source code analysis would be preferred. To this end, we implemented essentially two different types of detectors: The first type of detectors was responsible for recognizing relevant architectural elements, like API Interfaces, API Operations, and specific calls and invocations that establish interservice communication. We, therefore, refer to them as *Hot Spot Detectors*. The second type, *Invocation Detectors*, are responsible for tracking call chains between the various hot spots, thus creating the final graph structure representing our communication model. As the approach should be offered as a reusable component to be used e.g. in a continuous delivery pipeline of a project, based on the data from our case study reported in Sect. 4.1.2, we selected Approach 2 for the two additional cases. We developed reusable detectors that were not bound to any project-specific implementation, apart from some project-specific code in order to reduce the implementation effort (see below) (Table 3).

The first API model case study was conducted on two versions of the open-source *Lakeside Mutual*[7] project. This project realizes the architecture of a fictional insurance company and consists of several Java Spring-based backend microservices. To detect the relevant hot spots and invocations, we had to implement ten detectors in total. While these were specific to Java-based communication technologies, such as *SpringRestControllers*, *FeignClients* or *JmsTemplates*, their usage would a) not be restricted to any concrete project and b) could easily be adapted for other Java-based microservice implementations. The overall implementation for analyzing the Lakeside Mutual project took 431 lines of code (LoC) for Java-specific detectors and 373 lines of generic code to orchestrate the detection process and generate the model, resulting in a total implementation size of 804 LoC. Using our detectors on the Fall 2020 revision of the project,[8] we were able to identify 40 different API operations (35 synchronous and five asynchronous ones) and 16 interservice connectors between these operations. Based on the generated model, we were able to track two domain-based cycles in the system. Our detectors were also able to analyze the latest[9]. version of the Lakeside Mutual project without requiring any changes to our existing code base. The analysis revealed a slight decrease in synchronous API operations (to 29), and we could verify that all cyclic dependencies we had detected in the previous version had

[7] https://github.com/Microservice-API-Patterns/LakesideMutual.

[8] https://github.com/Microservice-API-Patterns/LakesideMutual/tree/spring-term-2020.

[9] https://github.com/Microservice-API-Patterns/LakesideMutual/commit/bdc6d30135149563c057dd30f21b7df68608c500.

been resolved. Although the architecture has undergone significant changes between the two revisions, this case study demonstrates that our detectors were still able to identify the relevant structural elements in both versions without adjustments, which underlines the reusability aspect of our approach.

In the second API model case study, we examined the communication structure of the *eShopOnContainers*[10] system, an open-source microservice reference implementation for the .NET technology stack. Compared to Lakeside Mutual project, this one uses a pure event-based asynchronous communication model for interservice communication. In addition, some service implementations pursue a domain-driven design approach, resulting in a more implicit invocation call chain within the services themselves. While writing generic reusable detectors to track these invocations would be possible, generically covering all of these cases would require a considerable amount of implementation effort. Therefore our detector implementation used some heuristics that were specifically tailored to the underlying project. Because of that, our implementation amounted to 162 lines of code for project-specific detectors and 181 lines of generic detectors that could also be reused for other C#-based microservice applications. Table 4 summarizes the implementation efforts of our case studies. Please note that due to the close syntactic relationship between Java and C#, the amount of language-specific detector code could be reduced by implementing some of the generic detectors in a language-agnostic way.

The two API model cases show that the detector-based approach is well suited for problem-specific scenarios, too. The effort for implementation and configuration is significantly lower than using language-specific parsers, especially if these language parsers have to be kept up-to-date. While the generic detector approach might require some upfront implementation, this work amortizes considerably soon if more than one project revision needs to be analyzed, as seen in our first case. The ability to combine both approaches in the second case in order to reduce implementation effort shows that the approach is flexible, while the two projects show that detectors can be used to retrieve the system architecture automatically, eliminating the need for manual maintenance of the architecture model.

In terms of the design requirements, our approach satisfies in both cases R1 and R2 (adapted to the given context, e.g. it does not need to cover the entire system architecture). R3 is not applicable due to the mostly homogeneous code basis of the relevant parts of the two projects examined, but the detectors could easily be (re)written so as to cover both Java and C#-based systems. R4 is not relevant, as there is no manually created architectural model. R5 is implicitly supported by the approach, but traceability is not used any further in the cases. R6 and R7 are of limited relevance due to the different context, i.e. the focus on a subset of the system, and the automatic detection of relevant elements only; but the overall coding effort required in LoC (and thus, implicitly, time) is a fraction of the project size, even when leaving aside the gains from detector reuse relative to the changes in the project implementation over time.

[10] https://github.com/dotnet-architecture/eShopOnContainers.

Table 4 Size comparison between detector implementations for two example case studies

	System Size (LoC)[1]	Generic reusable code(LoC)	Language-specific reusable detector code (LoC)	Project-specific detector code (LoC)	Total implementation (LoC)
Lakeside mutual	35.2K /35.4K[2]	373	431	–	**804**
eShopOn containers	81.4K	373	181	162	**716**

[1] All files except *json* configuration files, calculated with *cloc*: http://cloc.sourceforge.net/
[2] *Spring-Term 2020* branch/master branch *April 2021*

8 Threats to validity

It is important to consider the threats to validity during the design of the study to increase its validity. Wohlin et al. [36] distinguish four types: conclusion, internal, external, and construct validity. Internal validity address establishing a causal relationship between variables. It is not relevant for this study, as we do not aim to create a causal relationship between variables using statistical means.

Conclusion Validity Threats to the conclusion are concerned with issues that affect the ability to draw the right conclusions between the treatments and the outcome of the study. As we do not apply statistical testing, related sources to conclusion validity reported by [36] do not apply to our study. As data reported about the resulting design and the case study ground truth is collected semi-formally (i.e., contains qualitative elements), there is the risk that the researchers' background, development experience, and understanding influence the interpretations. This risk is reduced as the different research team members carefully reviewed the steps taken by the other researchers. The risk is further reduced by the researchers' deep background in both the microservices domain and architecture abstraction methods. Finally, we included industry experts as authors to further reduce the bias. To ensure the reliability of our measures, we used objective measurements such as precise time recording and lines of code. For both a subjective element remains: other developers might have needed a different amount of time or structured their lines of code differently. Also, a few minor aspects of our measurements are estimated such as the common code base of Approach 1 and Approach 2. As we only aim for the measurements to provide a very rough estimate (i.e., in orders of magnitude), we believe possible differences to be negligible. The choice to perform a so-called *mechanism experiment* [17], i.e. implement the architecture model abstraction prototypically for the case ourselves, might have negatively influenced the reliability of the treatment implementation for the reported measurements and the comparison between the two approaches. Regarding RQ1 and RQ2, both of which investigate whether and how a feasible design is possible, the reliability of the treatment implementation is given, as two feasible designs have been found. We do not claim that the found designs are the only possible designs or optimal.

Construct Validity Construct validity is concerned with obtaining the right measures and instruments for the phenomena being studied. Our study combines design science and case study research to minimize possible mono-method bias. There is a risk of a possible interaction between treatments, as the treatments in the two approaches are applied one after another. Again, this does not impact the research for RQ1 and RQ2. A learning effect may have impacted the comparison between the two approaches. However, we argue that our goal is only to understand the differences in orders of magnitude, and do not claim generalizability of the precise measures. Continuous comprehension support, i.e., the ability of our detectors to detect all changes in the implementation (addition, modification, removal) of connectors in every release version of the system, has been tested in Sect. 6.2. The only problem we anticipate would be in the case where major changes occur between releases, but this goes beyond continuous comprehension, and effectively requires a *de novo* architectural reconstruction. Additionally, the evolution aspect has been tested and validated in three systems already for the predecessor work [15] of the approach presented here.

External Validity Threats to external validity are conditions that limit our ability to generalize the results. Our research has been performed by researchers and not in an industrial setting. This might limit the generalizability to industrial practices. As our main line of research aims to find a feasible design in context, the threat appears negligible. For the approach comparisons and precise measurements, the threat is realistic for the reasons given above. In addition, in order to come up with a sound design, we applied substantial refactoring effort throughout the study, which may not reflect current common practice in industry. That is, the reported effort might be slightly higher than what a "quick and dirty" implementation in industry would actually require. On the other hand, no effort was needed for meetings, design sessions, deployment, and thus also not reported. This might substantially increase the relevant measurements when creating industry-grade solutions. As all this should not considerably impact the *relative* difference between Approach 1 and Approach 2, we believe the threat to be negligible for RQ3. Finally, the basic soundness of the approach has been tested on open-source case studies realized by industry practitioners. This way we have ensured that our approach is applicable in a realistic setting. The fact that the case study projects are merely demonstrators and not a production-ready systems might result in differences compared to actual industrial implementations. From our point of view, the risk is that real-life industry systems are usually larger and less well-structured than the system we have studied. From our experiences with industry systems in the microservice domain (we included industry experts in the author team to confirm this point), many current industry systems are realized in a similar fashion, and we are confident that a) the chosen system is a representative cross-cut of current practices in the microservice domain and b) that it applies a sufficient number of different technologies so as to be a representative of industrial polyglot systems. The risk that our methods might require additional engineering when applied to very large-scale systems remains, but this is a question of additional adaptation and coverage effort, and does not invalidate the basis of our approach. On the contrary, once a basic set of technologies has been covered and a sufficiently large library of detectors exists, and with a more automated method for detecting architecture changes so that the manual model maintenance is further reduced, we are confident that the savings in effort due to the reusability of the detectors will end up being much more pronounced in a continuously-evolved, large-scale system than in our case study.

9 Conclusions and future work

In this paper we have reported on a design-science study combined with case-study research for studying methods for comprehending the component architecture of highly polyglot systems, as exemplified by state-of-the-art microservices systems. With regard to **RQ1**, we conclude that our study revealed that the approach taken by [15] (discussed in Sect. 3) probably allows to design a highly accurate architecture abstraction, but that it involves considerable effort in a highly polyglot setting. By contrast, both of the semi-automatic, detector-based approaches developed in our study work well in the highly polyglot setting and fulfill all our requirements. We further observed that the reusable detectors from Approach 2 tend to enable cleaner

solutions: due to the detector specificity, similar detections sometimes led to different approaches in Approach 1, whereas in Approach 2—as reuse was already a goal of the design—the solution was provided by more generic, common code. Thus, Approach 2 makes it more unlikely that recurring code issues or possible defects stay undetected. Moreover, in a system with 11 different technologies, we managed to retrieve its component architecture with a reasonable effort (i.e., within a reasonably short period of time), with high hopes for high reusability and small deltas during future evolution of the same system. With regard to **RQ2**, all three approaches studied can support continuous comprehension well. The approach by Haitzer and Zdun is limited, as many parser technologies and grammars need continuous maintenance and testing effort; the two detector approaches developed as part of the present study work better in this regard. Approach 2 is superior to the one from Approach 1, as new or changed detection requirements need to be realized only once, in reusable detectors, and are then applied automatically for all projects using those same detectors. Approach 1, in contrast, would require searching for all related custom detectors and changing them individually. As a result, the reusable detector approach also seems to be slightly better suited for RQ2, as it requires a lot of discipline and refactoring to reach the same quality of evolution stability in Approach 1 as is built into Approach 2. For **RQ3** we have compared our two approaches in terms of time (effort), using time recording, and lines of code measurements. The results indicate that creating new reusable detectors for a single project requires substantially higher time (effort) for Approach 2 than for Approach 1. Creating the case-specific code requires significantly less time (effort) for Approach 2 than for Approach 1. Our data indicates there is a break-even point where the reusable detector method pays off. A very rough estimation, based on the averages in our case study data, indicates that the break even-point for effort is reached at about 6 uses of a reusable detector. For the lines of code measurements, it is reached after 8 uses. This does not consider the extra effort needed in Approach 1 compared to Approach 2 during software evolution; to cover those, further studies of an evolving software system would be needed. Based on all data and observations, we thus would recommend the reusable detector approach, unless a project is certain to use all detectors a very few times at maximum. We have also tested the adaptability of the same approach to different tasks, which are well suited to be used in combination with the architecture abstraction detectors, potentially greatly reducing the already small manual overhead required by our method.

As future research we plan to further investigate the methods reported in this paper in an industry context, and for other kinds of models than component models. We further plan to exploit the traceability links provided through our method in various research approaches. It would also be interesting to investigate whether a more precise prediction of required time and efforts is possible.

Acknowledgements This work was supported by: FFG (Austrian Research Promotion Agency) project DECO, No. 864707; FWF (Austrian Science Fund) project API-ACE: I 4268; FWF (Austrian Science Fund) project IAC^2: I 4731-N. Our work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952647 (AssureMOSS project).

Funding Open access funding provided by University of Vienna.

References

1. Newman S (2015) Building microservices: designing fine-grained systems. O'Reilly, Sebastopol
2. Lewis J, Fowler M (2004) Microservices: a definition of this new architectural term. http://martinfowler.com/articles/microservices.html
3. Zimmermann O (2017) Microservices tenets. Computer Science - Research and Development 32(3):301–310
4. Pautasso C, Zimmermann O, Amundsen M, Lewis J, Josuttis N (2017) Microservices in practice, part 1: Reality check and service design. IEEE Software 34(1):91–98
5. Hasselbring W, Steinacker G (2017) Microservice architectures for scalability, agility and reliability in e-commerce. In: Proceedings 2017 IEEE international conference on software architecture workshops (ICSAW), IEEE, Gothenburg, Sweden, pp. 243–246, https://doi.org/10.1109/ICSAW.2017.11
6. Pahl C, Jamshidi P (2016) Microservices: a systematic mapping study. In: 6th International conference on cloud computing and services science, pp. 137–146
7. Pautasso C, Wilde E (2009) Why is the web loosely coupled?: a multi-faceted metric for service design. In: 18th International conference on world wide web, ACM, pp. 911–920
8. Richardson C (2017) A pattern language for microservices http://microservices.io/patterns/index.html
9. Zimmermann O, Gschwind T, Küster J, Leymann F, Schuster N (2007) Reusable architectural decision models for enterprise application development. In: International conference on the quality of software architectures, Springer, pp. 15–32
10. Perry DE, Wolf AL (1992) Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17(4):40–52
11. Zimmermann O, Zdun U, Gschwind T, Leymann F (2008) Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method. In: Software architecture, 2008. WICSA 2008. Seventh working IEEE/IFIP conference on, pp. 157–166, https://doi.org/10.1109/WICSA.2008.19
12. Ducasse S, Pollet D (2009) Software architecture reconstruction: A process-oriented taxonomy. IEEE Transactions on Software Engineering 35(4):573–591
13. Murphy GC, Notkin D, Sullivan K (1995). Software reflexion models: bridging the gap between source and high-level models, In: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering, ACM, New York, NY, USA, SIGSOFT '95, pp. 18–28
14. Mens K, Mens T, Wermelinger M (2002). Maintaining software through intentional source-code views, In: Proceedings of the 14th international conference on software engineering and knowledge engineering, ACM, New York, NY, USA, SEKE '02, pp. 289–296
15. Haitzer T, Zdun U (2014) Semi-automated architectural abstraction specifications for supporting software evolution. Science of Computer Programming 90:135–160
16. Garcia J, Ivkovic I, Medvidovic N (2013). A comparative analysis of software architecture recovery techniques, In: Proceedings of the 28th IEEE/ACM international conference on automated software engineering, IEEE Press, Piscataway, NJ, USA, ASE'13, pp. 486–496
17. Wieringa RJ (2014) Design science methodology for information systems and software engineering. Springer
18. Alshuqayran N, Ali N, Evans R (2016) A systematic mapping study in microservice architecture. In: IEEE 9th International conference on service-oriented computing and applications (SOCA), IEEE, pp 44–51

19. Francesco PD, Malavolta I, Lago P (2017) Research on architecting microservices: trends, focus, and potential for industrial adoption. In: 2017 IEEE International conference on software architecture (ICSA), pp. 21–30. https://doi.org/10.1109/ICSA.2017.24
20. Granchelli G, Cardarelli M, Francesco PD, Malavolta I, Iovino L, Salle AD (2017) Towards recovering the software architecture of microservice-based systems. In: 2017 IEEE International conference on software architecture workshops (ICSAW), pp. 46–53
21. Alshuqayran N, Ali N, Evans R (2018) Towards micro service architecture recovery: an empirical study. In: 2018 IEEE International conference on software architecture (ICSA), pp 47–4709. https://doi.org/10.1109/ICSA.2018.00014
22. Vianden M, Lichter H, Steffens A (2014) Experience on a microservice-based reference architecture for measurement systems. In: 2014 21st Asia-Pacific software engineering conference, vol 1, pp. 183–190, https://doi.org/10.1109/APSEC.2014.37
23. Rademacher F, Sachweh S, Zündorf A (2019) Aspect-oriented modeling of technology heterogeneity in microservice architecture. In: 2019 IEEE International conference on software architecture (ICSA), pp. 21–30. https://doi.org/10.1109/ICSA.2019.00011
24. von Detten M, Becker S (2011) Combining clustering and pattern detection for the reengineering of component-based software systems. In: Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS, ACM, New York, NY, USA, QoSA-ISARCS '11, pp. 23–32
25. Corazza A, Di Martino S, Scanniello G (2010). A probabilistic based approach towards software system clustering, In: Proceedings of the 2010 14th European conference on software maintenance and reengineering, IEEE Computer Society, Washington, DC, USA, CSMR '10, pp 88–96
26. Sartipi K (2003). Software architecture recovery based on pattern matching, In: Proceedings of the international conference on software maintenance, IEEE Computer Society, Washington, DC, USA, ICSM '03, pp 293-
27. Stoermer C, Rowe A, O'Brien L, Verhoef C (2006) Model-centric software architecture reconstruction. Softw Pract Exp 36(4):333–363. https://doi.org/10.1002/spe.699
28. Murphy G, Notkin D, Sullivan K (2001) Software reflexion models: bridging the gap between design and implementation. IEEE Trans Softw Eng 27(4):364–380. https://doi.org/10.1109/32.917525
29. Knodel J, Muthig D, Naab M, Lindvall M (2006) Static evaluation of software architectures. In: Software maintenance and reengineering, European conference on pp. 279–294
30. Ganesan D, Lindvall M (2014) Adam: External dependency-driven architecture discovery and analysis of quality attributes. ACM Trans Softw Eng Methodol 23(2):17:1-17:51. https://doi.org/10.1145/2529998
31. Fittkau F, Krause A, Hasselbring W (2017) Software Landscape and Application Visualization for System Comprehension with ExplorViz. Information and Software Technology 87:259–277. https://doi.org/10.1016/j.infsof.2016.07.004
32. Haitzer T, Navarro E, Zdun U (2017) Reconciling software architecture and source code in support of software evolution. Journal of Systems and Software 123:119–144
33. Boehm B, Clark B, Horowitz E, Westland C, Madachy R, Selby R (1995) COCOMO 2.0. Ann Softw Eng 1(1):1–24
34. McConnell S (2004) Code Complete: A Practical Handbook of Software Construction, 2nd edn. Microsoft Press, Redmond
35. Wolff E (2016) Microservices: flexible software architecture. Addison-Wesley Professional
36. Wohlin C, Runeson P, Hoest M, Ohlsson MC, Regnell B, Wesslen A (2012) Experimentation in Software Engineering. Springer