Master's thesis

Master's Programme in Computer Science

Light-weight method for detecting API breakages in microservice architectures

Sami Lahtinen

August 20, 2022

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Sami Lahtinen

Työn nimi — Arbetets titel — Title

Light-weight method for detecting API breakages in microservice architectures

Ohjaajat — Handledare — Supervisors

Dr. A-P. Tuovinen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	August 20, 2022	51 pages, 14 appendix pages

Tiivistelmä — Referat — Abstract

Background: The splitting of functionality into multiple inter-communicating services has created a need for managing the APIs that these services are using to talk to each other. They also present an easy avenue for faults to be introduced into the system as these services are updated over time, especially in the absence of extensive testing, such as in rapid prototyping contexts. **Aims:** The study aimed to find a light-weight method for detecting API breakages between services, which requires as little manual labor from developers as possible. **Method:** The method used for the study was design science research focused around the iterative development and validation of the method through using it in synthetic and practical use-cases. **Results:** The study identified the possibility of using self-documenting services and machine-readable API documentation as a means to automatically detect API breakages either via naive or more complex approaches, with complex approaches providing more granular fault detection and ability to create dependency graphs between services. **Conclusion:** Use of more automation seems a viable approach to detecting faults in network-based communication between services. With further study, these approaches could be developed into developer-friendly systems, which allow not only automated fault detection, but also visual impact analysis for complex architectures spanning multiple services.

ACM Computing Classification System (CCS)

Software and its engineering → Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging

Software and its engineering → Software notation and tools → Software maintenance tools

Information systems → World Wide Web → Web services → RESTful web services

Avainsanat — Nyckelord — Keywords

web services, testing, RESTful, microservices

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction **1**
1.1 Research context . . . 1
1.2 Objectives and methodology of the thesis . . . 2
1.3 How this study is organized . . . 3

2 Background **5**
2.1 Fault detection techniques in software engineering . . . 5
2.1.1 Software testing . . . 5
2.1.2 Static analysis . . . 6
2.1.3 Continuous Integration and Continuous Deployment . . . 7
2.2 Microservice architectures . . . 7
2.2.1 Inter-service communication in microservice architectures . . . 9
2.2.2 Testing of microservice architectures . . . 11
2.3 REST APIs and their specification . . . 12
2.3.1 OpenAPI Specification . . . 12
2.3.2 Tools for working with OpenAPI documents . . . 13

3 Methods **14**
3.1 Research questions . . . 14
3.2 Specification of the design artifact . . . 14
3.3 Stages of the study . . . 16
3.3.1 Discovery of applicable techniques . . . 16
3.3.2 Implementation phase . . . 17
3.3.3 Validation . . . 17

4 Results **19**
4.1 Description of the artifacts . . . 19
4.1.1 Bouncer . . . 19
4.1.2 System architecture . . . 21

4.1.3 Using Bouncer for tracking API changes . . . 22
4.2 Validation . . . 25
4.2.1 Testing an internal SDK . . . 25
4.2.2 Validation against a test microservice architecture . . . 27
4.3 Adopting the system into software development process . . . 29
4.4 Which kinds of API breakages could be detected . . . 31
4.4.1 Altered parameters or request body . . . 31
4.4.2 Altered response . . . 32
4.4.3 Changed HTTP method . . . 32
4.4.4 Changed URL path . . . 32
4.4.5 False positives . . . 33
4.4.6 False negatives . . . 33
4.5 Application of the system to API breakage detection . . . 34
4.6 Generalizability of the system . . . 35

5 Discussion **37**
5.1 Applying the system more generally to software engineering . . . 37
5.1.1 Utilizing code generation and client libraries . . . 37
5.1.2 Direct microservice communication versus event-based communication 38
5.1.3 Organization-level adoption . . . 39
5.2 Related work . . . 40
5.2.1 Contract-first development . . . 40
5.2.2 Other inter-service interface definition languages . . . 41
5.2.3 Comprehensive automatic test generation . . . 42
5.2.4 Language-based approach to microservice development . . . 42
5.3 Future work . . . 43
5.4 Threats to validity . . . 44

6 Conclusions **46**

Bibliography **48**

A Example OpenAPI specification

B Bouncer Client source code

C Bouncer Client diff listing

D Bouncer test coverage

1 Introduction

In programming, one of the simplest levels or error detection in a given program is checking if the function calls and object references in the program match with other declarations in the program. Certain languages also allow checking that those functions and objects are used in a correct way by utilizing stricter typing and type checking. Often all of this is done even before the software is ever executed and thus these kinds of errors relatively rarely end up in actual software deployments.

However, modern software doesn't just make function calls to other parts of the same software, and the software system is often split up into a swarm of distributed services, such as interconnected microservices that talk to each other through network-exposed APIs. These services can change independently from each other and these changes may end up being reflected in API boundary, creating incompatibilities. Especially APIs in their early state of evolution are prone to changing significantly (Espinha et al., 2015). However, it is also noted in (Espinha et al., 2015) that maintaining multiple versions of an API on the provider side for extended period of time is also expensive, and is thus also discouraged.

Because of the decoupled nature of the inter-communicating services, web API compatibility is a dynamic concern and needs to be assessed continuously. Often this is achieved by manual testing and creation of comprehensive test suites and staging environments, that combined will reveal API incompatibilities between the running services.

This approach is no doubt the correct one for production-quality software, but this level of testing can be prohibitively expensive for software that is comparatively short-lived and doesn't have strict quality requirements.

1.1 Research context

At Nokia's Advanced Technology Group (ATG) we are focused on building exactly this kind of software. We produce technology evaluations, prototypes and proof-of-concepts that are primarily intended to be built quickly and only to the necessary quality and scope to demonstrate the viability of the technical approach, according to the principles

of prototyping (Budde and Zullighoven, 1990). This creates a software engineering optimization problem, where we need to apply just enough software engineering best practices to maintain decent velocity, but at the same time strip out any excess work that slows us down.

However, faults introduced to the system also end up slowing us down. Time spent on debugging a prototype to make it functional is time not spent on researching the relevant topic or working on other prototypes. Therefore we would like to spot as many critical issues with as little work as possible and as early as possible.

This means that there is demand for an automated system that allows us to identify mistakes in sending or receiving data from API endpoints or trying to use an API endpoint that no longer exists, similar to how compilers and linters can detect type and syntax errors. While this kind of an automated system for detecting API breakages may not provide as comprehensive of an error detection system as a full suite of integration tests, it should hopefully detect at least simple cases such as an API endpoint being renamed, deleted or potentially its arguments changing shape. With this sort of a system in place, we should be able to more quickly detect and react to inter- service communication defects and identify the services that need to be changed to facilitate certain API changes. When applied to a continuous integration pipeline, it should then be able to give us a fair amount of confidence that the various services that constitute our software system have a common understanding of the inter-service communication channels.

ATG also has a significant trainee presence and many times these trainees operate in autonomous groups without constant monitoring by a permanent ATG member. Therefore the system should be such, that it can be taught to a undergraduate student and deployed once without significant need for maintenance later. In an optimal scenario, the system would be simple enough that the trainees can simply read the documentation, deploy the software themselves and then begin applying the system to a new project without any intervention by a specialist.

1.2 Objectives and methodology of the thesis

The main goal of the thesis was to implement an automated system that can be used at Advanced Technology Group to give programmers tools to identify breakages in HTTP APIs with a focus on REST APIs in particular.

In order to accomplish that task, design science research was picked as the most suitable research methodology (Wieringa, 2010). Design science research focuses on the design and implementation of research artifacts aimed at solving a specific problem relevant to the research context. This approach matches the purpose and existing methods of work used at Advanced Technology Group and was deemed to provide the most immediate benefit to the team. The specifics of the methodology and activities relevant to it are covered in more detail in Chapter 3.

The creation of the automated system for API breakage detection involved identifying the specific properties of the problem, carrying out the implementation work of the system and then validating the solution against various example cases of API breakages in order to test out the solution's effectiveness.

1.3 How this study is organized

The study is organized in a way that generally matches the phases of the study from beginning to the end, starting with the background information and theory and then going through methodology for carrying out the design science research and ending with the results of the artifact construction and analysis of its performance in solving the problem set.

Chapter 2 establishes the background upon which the study was built. It covers the terminology, technologies and methods relevant to the research problem and covers fault detection approaches in software engineering, a general overview of microservice architectures and how REST APIs function and are specified.

Chapter 3 covers the methodology used to conduct the study. The key aspects in the chapter are the definition of the goals of the study and the research questions which the study aims to answer, along with the definitions of the requirements placed on the artifacts from the study. The chapter also details the phases of the study and the activities relevant to each phase.

Chapter 4 goes over the results from the study and the description of the implemented artifacts and their function. It also contains the results from the selected validation cases for these artifacts and provides answers to the research questions set in Chapter 2.

Chapter 5 contains introspection on the context of the study in the scientific field and its relation to other studies, which weren't used directly for the purposes of the study but

which are useful at putting this study and its results in the proper context. It also aims to consider the results of the study in terms of scientific validity and rigor, as well as the more general implications of these results.

Chapter 6 provides conclusions to the study and summarizes the overall results.

2 Background

This chapter goes over background information relevant to detecting faults in the usage of REST APIs. Different types of fault detection techniques are detailed and their benefits and drawbacks are compared. After this, the focus is shifted onto how microservice architectures function and how REST API endpoints are defined using OpenAPI Specification. The material covered in this chapter forms the necessary background for the contexts in which REST API breakages may take place and that tools and techniques are available to address them.

2.1 Fault detection techniques in software engineering

There are various techniques for detecting faults in software projects. These techniques can be sorted into two categories, based on the relationship between the technique to the execution of the software. These two categories are *software testing* and *static analysis*.

Testing is based on executing parts of the software and observing the effects of this execution for logical inconsistencies (Machado et al., 2010). This approach may require significant amounts of the software to be executed in order to achieve required test coverage.

In static analysis the possible faults are detected through analyzing the software as logical statements and their relationships. Static analysis therefore does not require significant, if any, parts of the software to be executed.

Often software projects will use both approaches to varying extent in detecting software faults. They are not mutually exclusive, but instead complementary parts of the complete fault detection process.

2.1.1 Software testing

Software testing is based on observing the software in action and then recording deviations from the expected results in order to detect faults. This method ranges from executing

automated unit tests to manual testing done by quality assurance personnel.

In the case of black box and white box testing, the procedure involves the creation of test cases with documented expected behaviour of the system (Vincenzi et al., 2010). The test cases are then run and the results compared with expected behaviour. The key aspect is that for this type of method to reveal faults, large numbers of test cases need to be created and they must be designed to exercise large parts of the system to reach desired test coverage. However, at the same time, tests need to have suitable granularity since while very broad tests may help reveal the existence of faults, they may make locating the faults difficult.

In white box testing the test cases would likely be defined in the form of unit and integration tests, which can be automatically executed and which provide good code-level granularity in the case of detected faults. Black box testing may also utilize automation even in the case of end-to-end testing through use of tools like Robot or Cucumber paired with Selenium. However, in both of these processes a test suite of test cases must exist.

2.1.2 Static analysis

Static analysis is a technique of analysing software without running it by looking at it in the abstract to detect patterns, which appear to violate good programming practices (Ayewah et al., 2008). Type checking and detection of undefined variables are common static analysis operations provided by compilers and linters. Some compilers, such as the compiler for the Rust programming language (Pearce, 2021), perform some level of static analysis to detect typical programmer mistakes and even report outright errors if such faults are detected. Additionally static analysis can be performed with a separate tool, such as FindBugs, which focuses on coding defects (Ayewah et al., 2008) or Pixy, which looks for cross-site scripting vulnerabilities in PHP web applications (Jovanovic et al., 2006).

The benefit of static analysis is that it does not require a test suite to exercise the software, which makes its use comparatively inexpensive (Zheng et al., 2006). However, static analysis can only detect the kinds of patterns it has been taught about and the analysis may contain significant amounts of false positives due to static analysis typically being more conservative. Studies have shown false positive rates of around 50% (Zheng et al., 2006) (Jovanovic et al., 2006). This increases the work needed to use static analysis effectively, since addressing a fault may require additional work to determine, whether the

detected issue is a real concern or not. High rates of false positives can also reduce trust in the static analysis tools, leading to poor utilization.

If the amount of false positives is not too overwhelming, then static analysis provides a useful, light-weight tool to detect faults in software due to its low barrier for initial adoption and ease of automation.

2.1.3 Continuous Integration and Continuous Deployment

Continuous Integration (CI) is a software engineering practice where changes from multiple developers working on the same project are combined together (integrated) in a centralized source code repository frequently and the project is automatically built and tests executed (Meyer, 2014). This helps integrate changes to the project faster and provides a way to detect issues introduced by the changes. The CI system can also provide reporting of whether the build was successful, with the possibility to notify relevant developers of issues immediately (Shahin et al., 2017).

Use of CI tooling by itself does not uncover faults, except potentially in poor build environment documentation, but it works as an automated checklist that can execute automated tests reliably in response to developer activity. It also acts as a centralized information source about the current state of the software in terms of automated test suite success. The fault detection still relies on comprehensive test suites or static analysis tools, but automating the execution of these tools ensures they are used consistently on each change made to the the software system.

Continous Deployment (CD) is a continuation of the CI mindset, where in addition to automatically integrating and testing changes the software is also deployed automatically upon successful build and test phase (Shahin et al., 2017). This improves the turnaround time from a change being implemented to it being deployed. It also reduces the human labour required in the deployment process.

2.2 Microservice architectures

In a microservice architecture (MSA), the functionality of the application is split into small blocks called microservices, which are separate programs that are independently deployable and which can vary in implementation technology from one service to another (Dragoni et al., 2017).

The most important benefits in the context of Advanced Technology Group are the ability to separate parts of the system and use service-specific implementation technology. The former makes it possible to reuse parts of the system and the latter allows potentially multiple technology evaluations to take place in the context of a single project. Microservice architecture also provides the potential benefit of being able to split up work more easily between multiple teams.

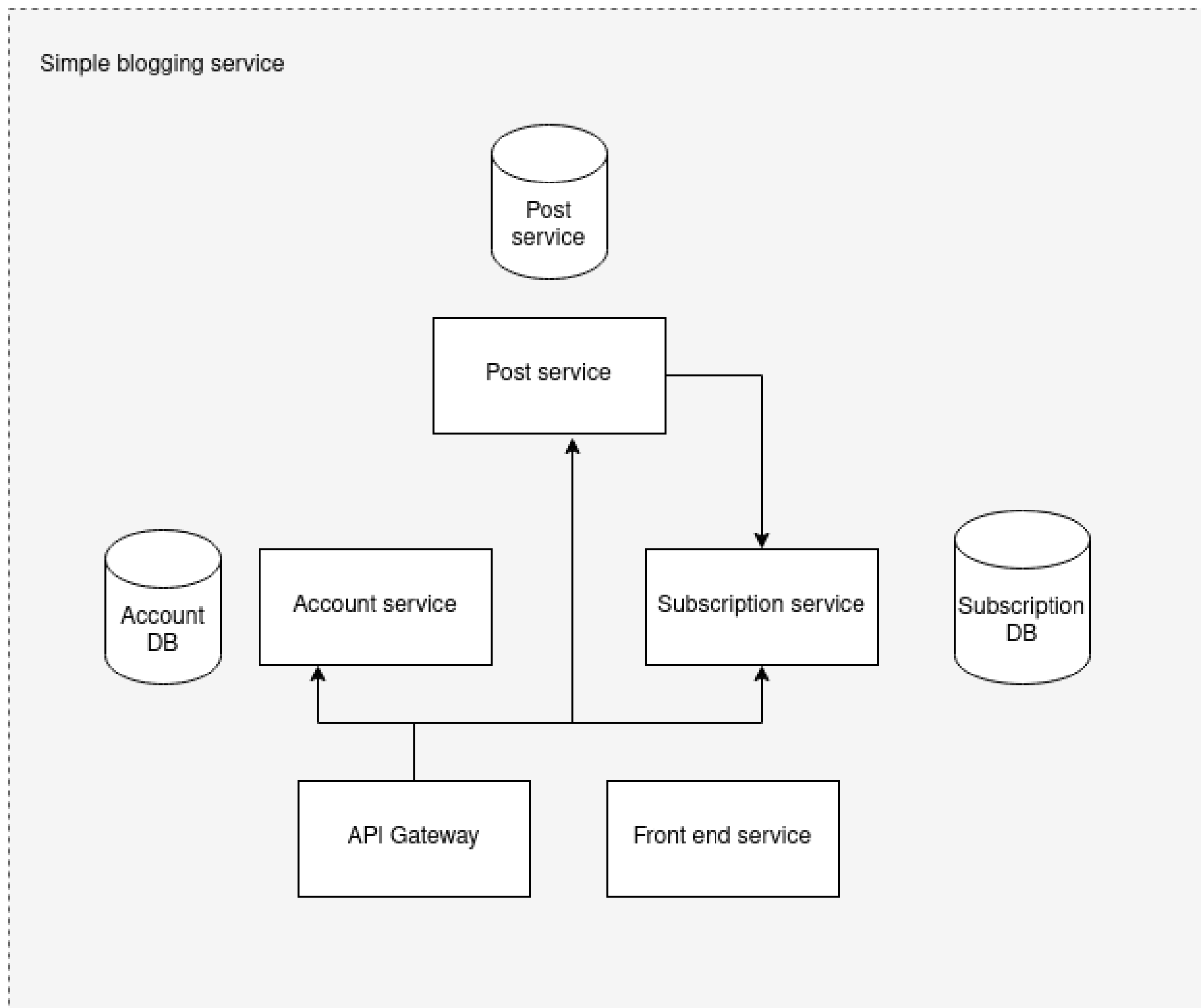

Figure 2.1: Example microservice architecture for a blogging service

Figure 2.1 shows an example microservice architecture for a simple blogging system, where user account functionality, blog posting functionality and subcribing to other blogs have been separated into individual services. The API gateway provides an access point for the APIs of each service. Additionally, the Post service talks directly to the Subscription service to update users' subscription feeds.

However, since microservices are independent of one another and because they implement

only a part of the complete system, they must cooperate in order to do larger tasks. The services cannot just call methods or access variables, so they must share and transform data using various communication protocols (Pacheco, 2018, p.21-23). The services must expose public interfaces, through which other services can gain access to the functionality of the service being called. And as with all communication protocols, the caller and callee must share a common understanding of how to communicate with each other. The emphasis on communication and the loose coupling between services makes the public interface of the service a difficult thing to change.

2.2.1 Inter-service communication in microservice architectures

Microservices have multiple communication channels available to them. Unlike in centrally orchestrated service-oriented architecture (SOA), the microservice architecture does not have a central integration layer, through which communication flows from one service to another (Cerny et al., 2018). Instead services are decentralized and can define multiple sets of communication channels between each other and to the outside world.

This means that communication in a microservice architecture can take many forms. Microservices can communicate directly with each other using a protocol like REST or decouple inter-service communication by using a publisher-subscriber message queue like Apache Kafka.

This flexibility in communication does create some challenges, though. For each direct interaction between two microservices, the microservice initiating communication needs to know three things: where the other microservice is located, what is the endpoint for accomplishing a specific task and what communication protocol is used in that endpoint. The communication protocol itself also involves knowing what data needs to be sent and how the responses from that endpoint are supposed to be interpreted.

Service discovery can be handled in multiple ways. The architecture can define a service registry, which keeps track of where a specific service is running. Other services can then query the service registry to acquire information for initiating communication to that service.

Endpoint and communication protocol information on the other hand is knowledge that the service is assumed to have already. Endpoints could theoretically also be stored in a registry, but endpoint selection ultimately falls on the service itself. The service is also unlikely to be reactive to changes in the communication protocol. These parts of

the communication problem just need to be solved correctly and faults identified through testing.

Relationships between interconnected microservices can be represented with the use of a service dependency graph (SDG). (Ma et al., 2019) describes a technique called GSMART which uses Java's reflection mechanism to find service endpoints and calls into the endpoints of other microservices and can then construct an SDG out of this information. This SDG can cover the entire microservice architecture and give a good overview the dependencies between all of the services. The SDG can then be used to track the potential effect of a change to a particular service based on the assumption that services dependent on that service may also need to change in response.

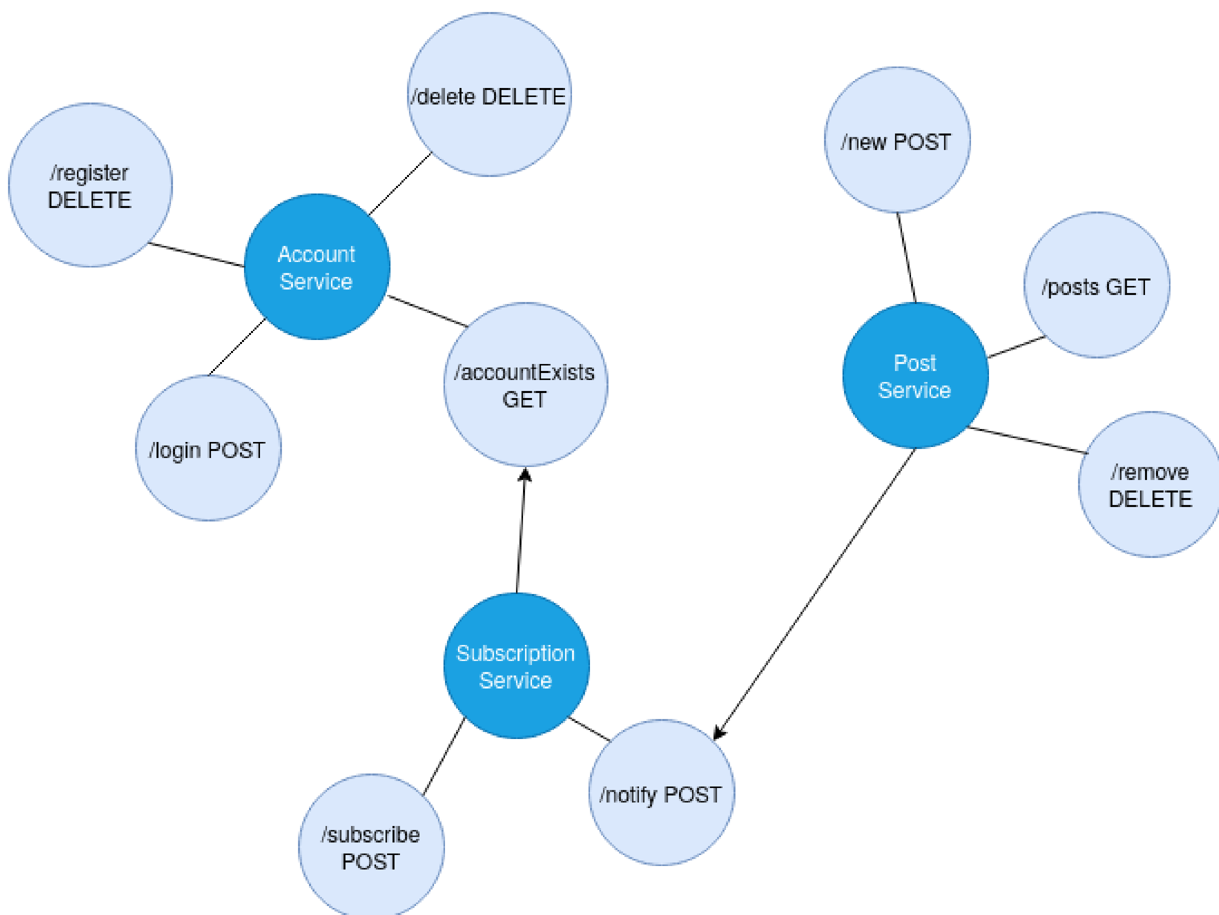

Figure 2.2: Example service dependency graph for a blogging service

Figure 2.2 shows an example SDG constructed from the blogging service example microservice architecture. Each of the core services is represented as a graph of associated API endpoints. Dependencies are established between the Subscription Service and the Account Service and between the Post Service and the Subscription Service and are repre-

sented with directional arrows. In this example we assume that the Post service needs to call an endpoint in the Subscription service to notify users of a new blog post having been published. The Subscription service on the other hand interacts with the Account service in order to verify that users can only subscribe to accounts that exist in on the platform.

2.2.2 Testing of microservice architectures

Microservice architectures make for an interesting challenge in terms of testing. Microservices need to be functionally consistent internally, but also especially in the larger context of the whole architecture. The separation between services means, that particular care needs to be placed on testing the interaction boundaries between services. Because the services are somewhat unaware of each other, the boundaries cannot easily be identified to be correct or incorrect.

Microservices themselves can be tested in isolation to ensure their internal behaviour matches the developers' expectations, but this only gives a limited understanding of the state and quality of the whole system. Even though requests into the microservice and responses from other services can be represented using mock objects (Spadini et al., 2017) that simulate the behaviour of real service implementations in specific cases, this doesn't by itself ensure the services are communicating correctly. Such mock objects may become outdated or their implementation may have been faulty from the start. The mock objects therefore have their own implementation and maintenance burden.

In order to run integration tests of the system, the test environment needs to have many or all of the services deployed and running at the same time, because only then can inter-service communication and behaviour be tested (Pacheco, 2018, p. 292-294). This makes the testing environment comparatively complicated, since the testing procedure will involve bringing up and tearing down the entire architecture. Full end-to-end tests can verify that entire use-cases are functional, but they also require the whole architecture to be running in order for those tests to be executed. In a continuous integration pipeline, it may for example become infeasible to run the entire system, so pipelines may need to deploy to some sort of a staging environment and then execute tests against that environment.

At the heart of the testing issue is also the matter of test cases. If the test cases do not adequately cover the whole of the microservice architecture, defects may appear in poorly tested areas of the system. Deriving exhaustive test cases for the system is time-consuming and they will also require maintenance as the system is developed and changed.

(Gazzola et al., 2020) describes a technique, in which test cases can be automatically generated from captured interactions with the system. This way the system can be used in practice through human testing and the interactions analyzed and saved. Then test cases are derived from this data and they can be played back against the service in a test environment. This is useful for regression testing of stateless microservices, but its utility is limited when dealing with stateful interactions or intentional changes in the behaviour of the service.

2.3 REST APIs and their specification

REST is an architectural style for representing state transfer, often for HTTP-based APIs, that defines a way to create, read, update and delete information through otherwise stateless interactions with a web service (Feng et al., 2009).

In practice, REST separates application and resource state from each other and provides access to the resource state via URI-based resource identification. The HTTP methods GET, POST, PUT and DELETE then allow the mutation of the resource state on the server. The API consumer is then responsible for how it alters its own application state based on the responses it receives from the accessing or modifying resource state.

The statelessness of REST requests ensures that requests are context-free, which means that the request by itself contains all of the state information needed to process the request. This means that it is possible to create a definition of correct communication procedure with a correctly implemented REST service, in which API requests are considered in isolation from each other.

2.3.1 OpenAPI Specification

A REST API description format called OpenAPI Specification (formerly Swagger) was created to facilitate API discovery and usage with a computer-readable API description format based on JSON and YAML (*OpenAPI Initiative* 2022). It allows REST API paths and their associated dynamic elements, such as resource identifiers and request and response bodies to be specified in a single document.

OpenAPI Specification is detailed enough to allow the generation of code, which talks to the specified REST endpoints (Albano and Nielsen, 2020). This also makes it a candidate for a property-based test generation strategy, where the test cases and associated data is

derived from the OpenAPI document for a particular service (Karlsson et al., 2020). An example OpenAPI document in provided in appendix A of this study.

These approaches make it possible to either ensure compatibility between an API provider and an API consumer either by ensuring the consumer is only using code that is generated to be compatible with the provider or by generating test cases from the API specification to ensure that the API provider does not unexpectedly deviate from the previous specification.

The OpenAPI Specification also helps us identify the exact moving parts that constitute these REST API endpoints (Karavisileiou et al., 2020).

Of specific interest to this thesis are the following OpenAPI properties:

- Paths (containing HTTP method and service endpoint path)
- Schemas (the shape of objects that can be sent to or received from endpoints)
- Responses (definition of message content for different HTTP responses)
- Parameters (parameters passed to endpoints via URI, cookies and HTTP headers)

The correct analysis of the OpenAPI documents and collecting the matching data from the API consumer side makes it possible to identify inconsistencies in communication automatically. In combination with a service dependency graph, invalid dependencies can be identified and reported.

2.3.2 Tools for working with OpenAPI documents

Various tools have been created to allow developers to work with OpenAPI specification files to accomplish various tasks. Tools like Swagger-Codegen or openapi-generator can be used to generate boilerplate code for API servers or full client libraries to APIs which have been adequately detailed using OpenAPI specifications.

Of additional interest are tools like openapi-diff, which can be used to analyze changes between two OpenAPI documents to determine the types of changes that have taken place and categorize them into non-breaking and breaking changes. It is worth noting that the openapi-diff tool comes in at least two different flavours: one developed in Java by the OpenAPI Tools organization (*GitHub - OpenAPITools/openapi-diff* 2022) and one developed in TypeScript by Atlassian (*openapi-diff* 2022). These will be referred to as “OpenAPI Tools openapi-diff” and “Atlassian openapi-diff” respectively for clarity.

3 Methods

The study was conducted using design science research methods (Wieringa, 2010). This means that during the study an artifact was created based on the needs of the stakeholders of the research context. This artifact was then validated against these stakeholder needs in order to assess the utility of the artifact when applied to the practicalities of the research context.

3.1 Research questions

Through the design of the artifact and its validation the study aimed to answer the following research questions:

- RQ1: How easily can the system be adopted into the existing software development process?
- RQ2: What kinds of API breakages can the system detect and which kinds of breakages go undetected?
- RQ3: Does the system help developers detect and fix API breakages when they occur?
- RQ4: Is this approach generalizable across frameworks and programming languages?

3.2 Specification of the design artifact

The aim of this study was to produce a system that can be used by fast-paced development teams for quickly identifying API breakages in REST APIs. The system would need to be able to collect API information from multiple web services and establish dependency relationships between them, so that a developer would be notified of a breaking change and then adapt the dependent services accordingly or roll back the breaking change.

Some of the more specific requirements were derived from the research context. The focus on REST APIs was decided on due to REST being an established standard and widely

known and understood in the context of the team. It also helped limit the scope of the study. Additional requirements were placed on the types of services the system would need to interact with. At the ATG team, a decision was made prior to the study to write web services in Python utilizing the FastAPI framework, since FastAPI provides automatic generation of OpenAPI documentation. In order for the system to provide the most utility, it was decided to focus the efforts on the team's established technology stack. Ability to use the system in a CI/CD pipeline was also deemed a necessary requirement, since CI/CD provides the automation the team requires for developer efficiency and reliable test execution.

The final list of requirements for the artifact was the following:

1. The system must automatically collect and keep track of service REST endpoints.
2. The system must be able to receive knowledge about endpoint consumers and establish API dependencies.
3. The system must be able to detect a broken dependency.
4. The system will be designed with web services written in Python and utilizing the FastAPI framework in mind, first and foremost.
5. The system must integrate into a CI/CD pipeline to provide automated protection against broken API dependencies.

For the requirements of the artifact to be comprehensive, it is important to also define what a dependency relationship means in this context and when dependencies are considered broken. In this study we considered an API to have a dependency on another if during the execution of any of its methods a call to the other service is required for correct or fully functional operation. In this study dependencies are not differentiated between optional and mandatory dependencies and instead every dependency relationship is considered necessary.

A dependency is considered broken if any of its API endpoints change in a way that would require dependent services to change in order to continue working correctly. These kinds of changes should be detected and reported to the developer.

In the REST context, we consider a broken dependency to have one or more of the following characteristics:

- HTTP method does not match
- URL path does not match
- URL parameters or query parameters don't match
- Request body does not match
- Response body does not match

It is worth noting that REST APIs may also change in ways, which do not require the dependent service to change. For example, the creation of a new API endpoint is not a breaking change to a service that does not yet use it. API endpoints may also declare new optional parameters which also wouldn't be considered a breaking change. These types of changes being detected as an API breakage is deemed to be a false positive. This kind of a result is incorrect, but false positives were considered less serious than an actual API breakage going undetected, which would be a false negative.

Similar to static analysis, the decision was made to make the system initially more conservative in its API breakage detection and then apply techniques to reduce the number of false positives. This decision allowed the system to start out simple and complexity to be built up, rather than spending a significant amount of time on the first implementation with uncertain results. The overall design process of the artifact followed an iterative model, where a part of the system was designed and then iterated on later to fine-tune the system and improve areas where deficiencies were discovered.

3.3 Stages of the study

The study was conducted in 3 stages, each of which informed the next one and guided the design of the system. The stages were not executed fully sequentially and certain activities from prior stages were also conducted later where necessary. For example, some implementation work was done in parallel with collecting existing literature on the topic. Validation phase also informed some implementation activities.

3.3.1 Discovery of applicable techniques

The first part of the study was an exploratory, non-systematic literature review to discover approaches and tools for fault detection and API compatibility checking. The results of

this review are documented in Chapter 2.

The main purpose of this stage was to identify viable routes to take for light-weight API breakage detection or to highlight issues with existing processes, which were deemed too heavy for our purposes. It also helped map out the specifics of the research problem and how prevalent the problem is in the field of software engineering. This laid the foundation of principles and system upon which the artifact could be designed.

3.3.2 Implementation phase

After a potential approach and system requirements were identified through the literature review, the implementation of the system was started.

The development was done iteratively and partially carried out during the validation phase in order to address requirements or limitations that only became clear at a later point in time.

Since the system being development was to be used for defect detection, the system required a higher standard for trustworthiness. Therefore a test-driven development methodology (Janzen and Saiedian, 2005) was applied to development activities. The development therefore happened in short cycles where expectations of a particular functionality were defined in a test case and then the minimal amount of code to make the test pass was written. In the end this resulted in a test suite of 47 separate test cases and test coverage of 97%. Appendix D contains a full test coverage report. This set suite was also useful in addressing later requirements, such as integration of openapi-diff tooling into the system.

3.3.3 Validation

The system was validated by applying it to both real and test projects to assess the ease of integrating the system to the existing development process. Through these tests the system's interference on the development process and ability to detect API breakages was assessed through both practical and synthetic scenarios.

The system was first applied to a real-world scenario, that represents a degenerate case of a single API provider and a single API consumer. This project was not a microservice architecture at all, but served as an example of the most primitive kind of application of the system. This project allowed us to assess the ease of integration to the CI/CD pipeline and testing the basic dependency analysis. It also served as the first checkpoint

for iterating on the design of the system.

Next, the system was applied to a more complex test project, in which an example application was developed using a microservice approach. The system was then applied to track the dependencies of all of the services part of this test project. The system was utilized from the start of the test project throughout the development of the system. During development, various types of synthetic API breakages were introduced to various services in order to test out the system's ability to detect and report the API breakages.

Results from both experiments were documented and then analyzed to determine the overall success of the system. The system was also opened up for feedback to colleagues to gather more qualitative information about the system.

4 Results

This chapter describes the artifacts that were developed during the study and the results from the validation cases performed on the artifacts. The first section gives an overview of the software system that was developed and then describes how it can be used in a software engineering process to detect API breakages. The second section then applies the artifacts to two validation cases in order to provide answers to the research questions posed in Chapter 3.

4.1 Description of the artifacts

During the study, two major artifacts were created, one being a software system for automatically detecting API breakages in REST APIs called Bouncer and the other being the methodology of how to apply it to a software engineering project.

Bouncer is a system designed to integrate into a software engineering project's CI/CD pipelines in order to collect OpenAPI specifications for the different components of the project and to execute dependency checks for API breakages between these components. Software components can declare their API dependencies by placing OpenAPI documents inside the component's source code repository, which Bouncer can compare to API specifications it has received previously.

4.1.1 Bouncer

Bouncer is a simple HTTP service and associated client script, which used together allow detection of API changes in REST APIs created using the FastAPI framework (*FastAPI* 2022). FastAPI is a web service framework for Python, which can be used for the creation of REST APIs. From the point of view of this study, it's main selling point is its ability to automatically generate OpenAPI specifications from source code, which makes it useful for Advanced Technology Groups's prototyping purposes and for use with Bouncer.

The Bouncer service exposes HTTP endpoints, which allow OpenAPI documents to be uploaded in JSON format and then stored in an SQLite database or compared to existing OpenAPI documents. Dependency checks are handled by simply sending a local version of

a particular service's OpenAPI specification to Bouncer, at which point Bouncer compares it to an OpenAPI specification it has received from the service in question. A discrepancy between these two documents is considered an API break, at which point Bouncer will respond with an error code to indicate a failed dependency check. Bouncer was originally designed to assume any difference between two OpenAPI documents as an indication of an API breakage, but later on openapi-diff was integrated into the system to provide more intelligent analysis. Different types of API breakages that can be detected by Bouncer and potential cases for false positives are covered in more detail in the Validation section.

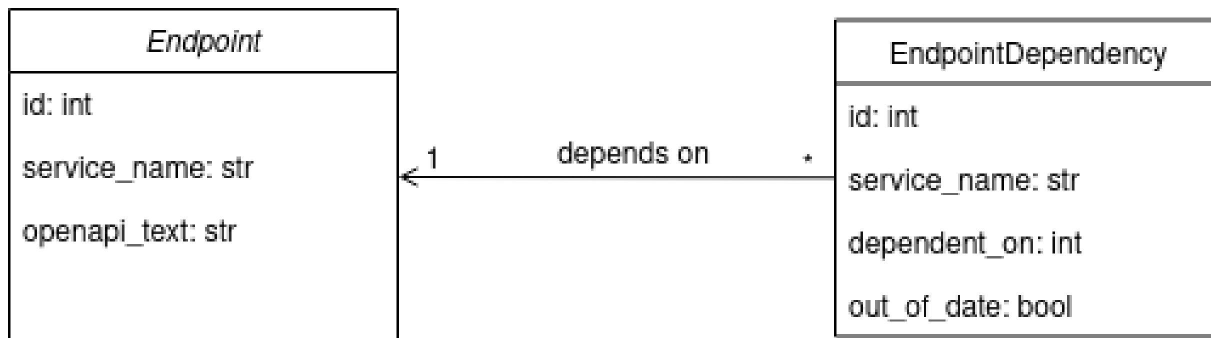

Figure 4.1: Bouncer class diagram

Figure 4.1 shows the core classes of Bouncer, namely the Endpoint and EndpoinDependency, which are used to execute dependency checks and to keep track of dependencies, which are out of date. When the OpenAPI specification for a service is changed, all endpoint dependencies on that service in the database are marked as out-of-date. Outdated dependencies are not immediately considered a breakage to avoid cyclical failures. Such a cyclical failure would be an endpoint provider not being able to update due to outdated dependencies and dependents not being able to update due to API breakage. Therefore the API breakage is only considered from API consumer side. However, being able to query which dependents are no longer up-to-date can be useful for reacting to API provider change.

The Bouncer client is intended to provide the Continuous Integration (CI) aspect of the system. It is designed to execute in a CI pipeline when changes to a particular service are made and then automatically extract the OpenAPI information out of the service and upload it to Bouncer along with the OpenAPI documents of all defined service dependencies. If the client receives information from the Bouncer service that a dependency check has failed, it will exit with an error code to stop the CI pipeline. The CI system can then alert the developers that an API breakage has occured. See Appendix B for the source

code to the Bouncer client.

4.1.2 System architecture

The Bouncer software exists in the context of a wider system, which consists of the in-development services, which Bouncer is used to track, the Bouncer service used to hold a centralized view of the state of various service APIs, the client application used to upload new information to the Bouncer service and the Continuous Integration (CI) system, on which the client is run regularly to perform API consistency checks.

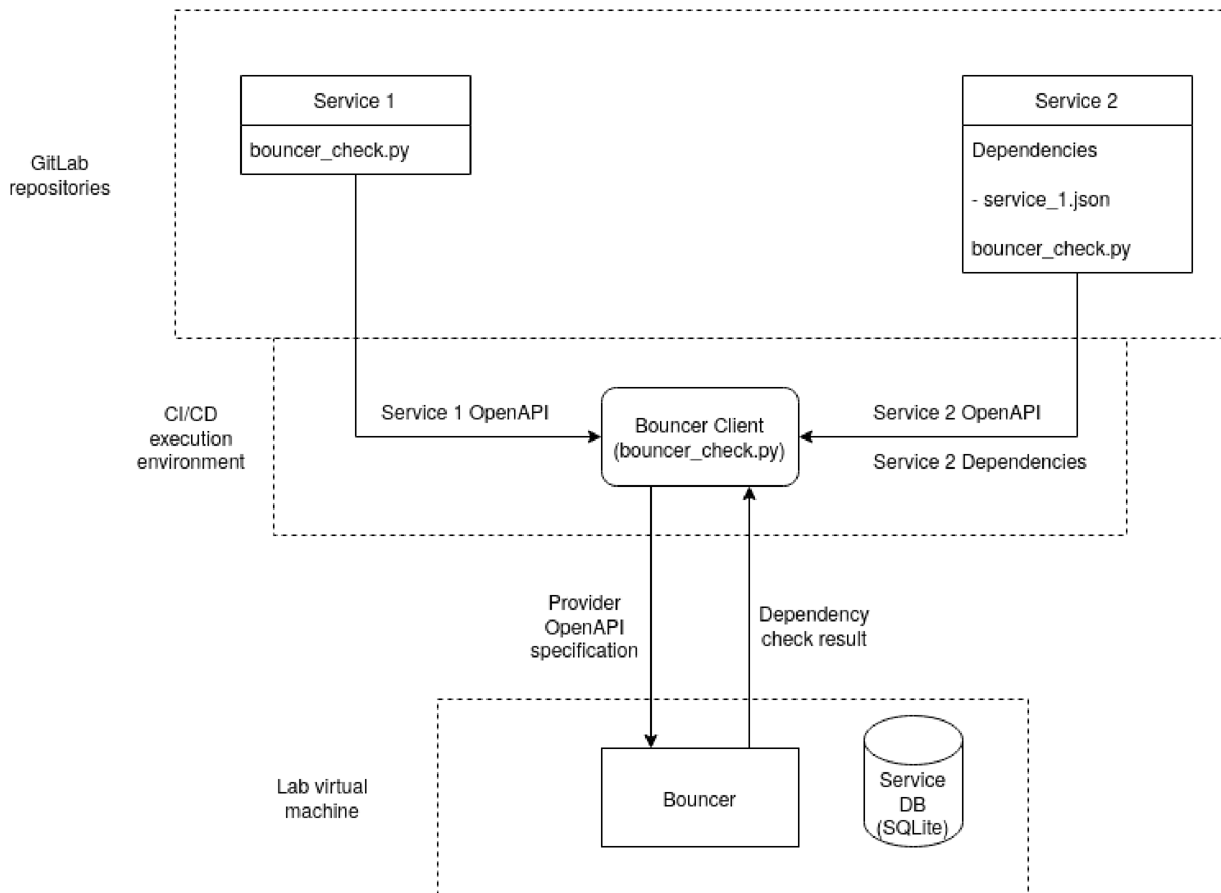

Figure 4.2: Bouncer system architecture

Figure 4.2 shows the relationship between the different parts of this system and where they exist in relation to each other. The key part of the system architecture that brings the system together is the Continuous Integration and Continuous Deployment pipeline (CI/CD), where the Bouncer Client can be executed where success and failure can be determined. The Bouncer Client can be considered an additional test that is run alongside

any other possible test suites of the in-development service under testing.

In the example architecture diagram, two services are defined: Service 1 and Service 2. Both of them produce their own OpenAPI specifications (referred to as the Provider OpenAPI specification). Additionally, Service 2 defines a dependency on API of Service 1 by including the OpenAPI specification as a JSON file in its source code repository. When Service 2 is updated and the Bouncer client executed in the CI/CD system, it will upload its dependency files along with its own Provider OpenAPI specification in order to execute dependency checks for API compatibility.

The selection of the CI/CD system is not a concern of Bouncer, as long as it can execute the Bouncer Client correctly. In the study, GitLab CI/CD was used, but could be substituted with other systems if needed.

4.1.3 Using Bouncer for tracking API changes

This section covers how Bouncer is set up for API breakage detection and change tracking in order to begin detecting breakages between two or more REST services.

Preliminary project requirements

While the Bouncer service is agnostic to the implementation technology of the APIs it tracks, provided an OpenAPI definition for the API can be written, the Bouncer client that was developed during the study only supports Python services using the FastAPI framework. This is due to the fact that while other web service frameworks with the ability to automatically generate OpenAPI information may exist, there isn't a simple way to target all of them simultaneously. A new client script would need to be written to support other frameworks or programming languages. Therefore it is assumed that all services being tested with Bouncer are FastAPI services.

The project is also assumed to have access to a CI/CD system, which will be able to run the Bouncer Client regularly and notify developers of pipeline failures. This is important for API breakages to not go unnoticed.

It is also assumed that an instance of Bouncer is deployed such that it is accessible from the CI/CD system and that it is kept running.

Tracking provider endpoints

Setting up Bouncer to track the API exposed by some service involves copying the Bouncer Client to the repository of the service being tested and configuring the parameters of the client. The client configuration lives inside the client script and consists of the following parameters:

- SERVICE_NAME: the name of the service being tested, used for setting up dependency links
- BOUNCER_INSTANCE: the URL to the accessible Bouncer instance
- DEPENDENCY_DIR: path to the directory, where OpenAPI documents for dependencies are stored

Additionally the script needs to be modified to import the FastAPI "app" from the appropriate module. This "app" variable provides the Bouncer client access to the OpenAPI information.

The CI/CD system needs to also be configured to execute the client script as part of the testing pipeline. Since most CI/CD systems should be compatible with Bouncer and because their configurations vary, the instructions have been omitted here.

Now running the client script either automatically via CI/CD or manually should result in the Bouncer service receiving up-to-date information about the endpoints provided by this service. Same steps can be taken with the source code repositories of other services.

Declaring API dependencies

Because Bouncer does not do analysis of the service's source code, it can only consider API dependencies that have been explicitly declared. Declaring an API dependency is performed by copying an OpenAPI document for the API that the service intends to use into the previously declared dependency folder.

When the Bouncer client is executed after dependencies have been added to the dependency folder, as part of the dependency check it will send each of those OpenAPI documents to the Bouncer service for compatibility analysis. If the Bouncer service reports any of the dependencies as broken, the Bouncer client will report an issue and halt.

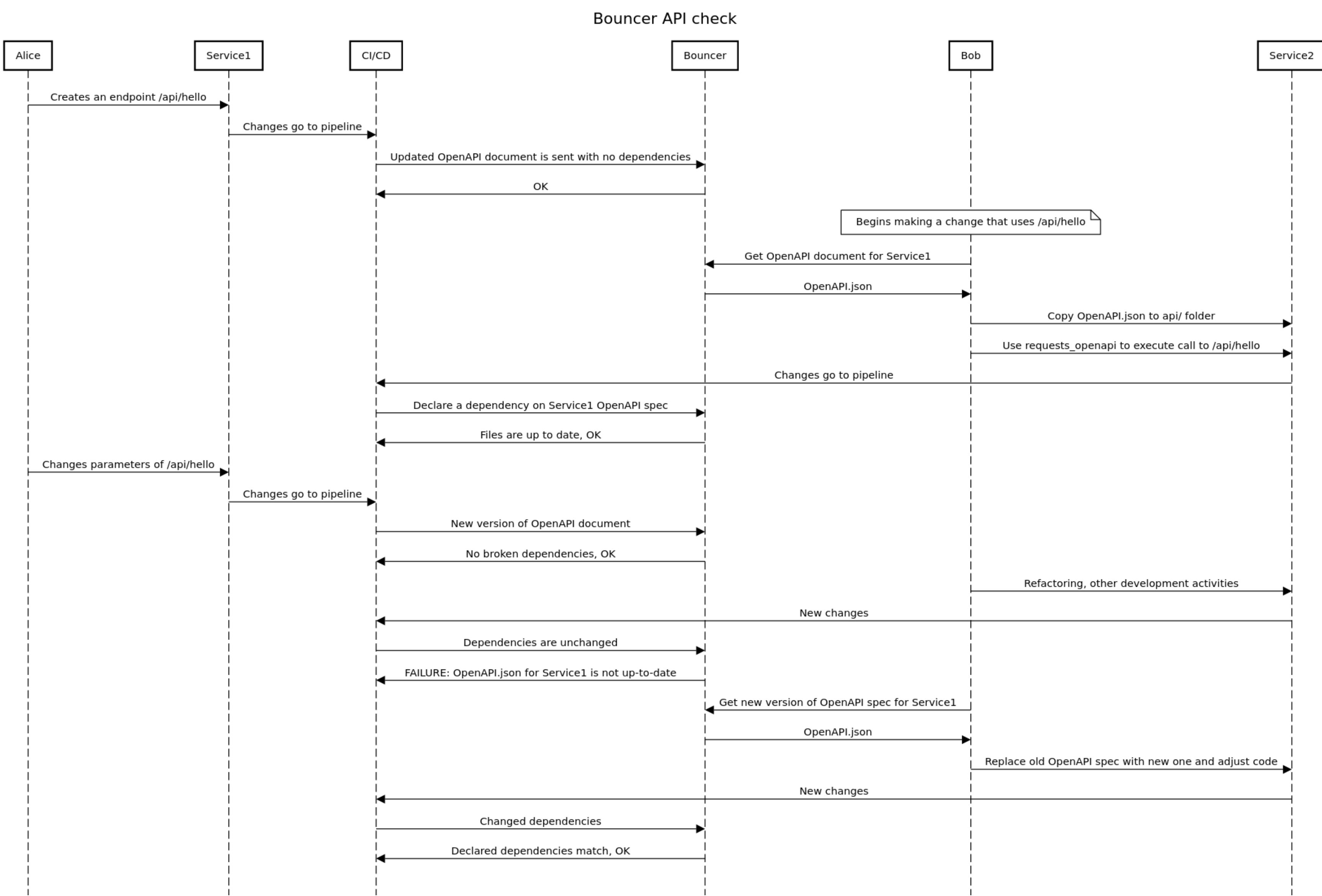

Figure 4.3: Example sequence of an API breakage resolution

Figure 4.3 shows an example sequence of actions, where an API is created by Alice for Service1. Bob then starts using the new API in Service2 by copying the OpenAPI document to Service2, which causes dependency checks to be run. We then assume Alice makes a breaking change to Service1, which gets reflected as the new version of the Service1 API. When Bob then makes changes to Service2, Bouncer will report an error due to depending on an incompatible version of Service1. The API breakage is resolved by redownloading the OpenAPI document and modifying Service2 to adapt to the API change.

4.2 Validation

4.2.1 Testing an internal SDK

During the study, the Bouncer system was tested in a practical scenario to monitor the API compatibility of an internal software development kit (SDK), which consumes a single API endpoint. This SDK was a Python library and as such provides no API endpoints of its own.

The SDK scenario was very different from the intended target for Bouncer, because neither the upstream API nor the SDK were FastAPI services. As such, this validation scenario did not provide validation for the typical Bouncer use case. However, it was a good opportunity to investigate RQ1 and the versatility of the Bouncer system.

Both the SDK and the API were designed before Bouncer existed, which means that none of the suggested workflows for effective Bouncer usage were being used at the beginning. The SDK made requests directly to the API and API compatibility was being tested using a suite of unit tests, some of which failed intermittently due to timing issues. This meant that integrating Bouncer was done into an existing development process from scratch.

In the SDK scenario, OpenAPI documentation for the upstream API was available in an online Git repository, but there was no access to modify the contents of this repository. Because of this, the Bouncer client was modified to fetch the OpenAPI document from the online repository instead of trying to get access to it via FastAPI. This meant that the CI process had the following steps:

1. Run unit tests.
2. Fetch OpenAPI document from upstream API repository and upload to Bouncer.

3. Upload the dependency OpenAPI document to Bouncer for dependency check.

We can see from the process that in this scenario the role of the Bouncer service is not actually necessary, since the dependency check could have been run inside the CI process without an external service. However, the Bouncer service was still used in this scenario to test how well it would integrate into the CI system.

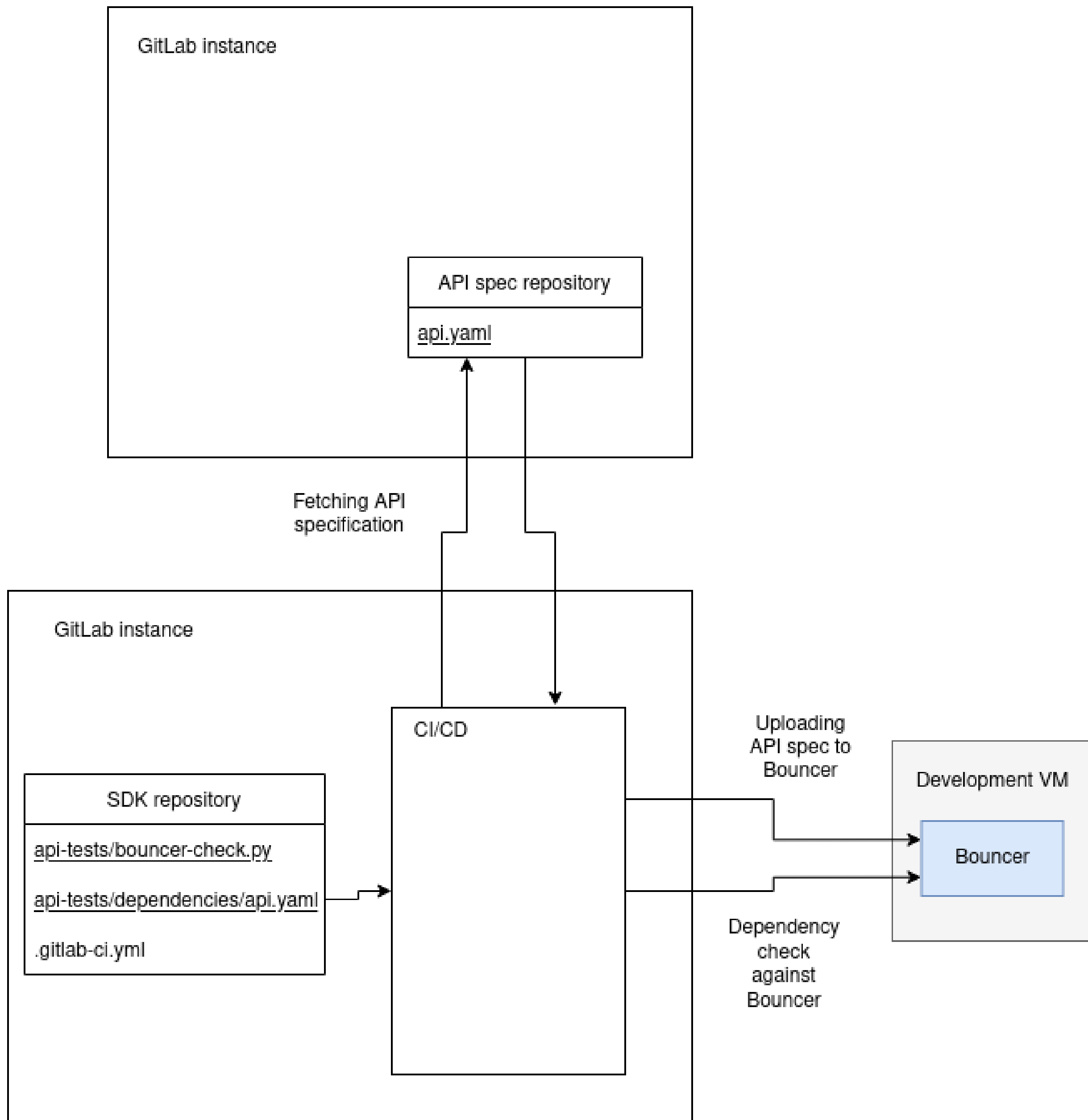

Figure 4.4: Bouncer SDK testing validation scenario overview

Figure 4.4 shows how the relationship between the SDK and API repositories and how

Bouncer integrates into the CI system.

In order to be operational, the Bouncer service had to be deployed in a location that was accessible from the CI system. For this purpose, a development VM was repurposed as a Bouncer server. Since the Bouncer service is very self-contained and relies on a SQLite database, deployment process was very smooth and after the initial deployment, the server-side software did not need to be modified. The only necessary configuration change was making the Bouncer service available on port 80 instead of 8000 due to organizational firewall practices.

The client script required some reworking due to the fact that neither the SDK nor upstream API are traditional FastAPI web services. So, the part related to acquiring API documentation from FastAPI was replaced with an HTTP request to a Git source code repository. The API specification was also written in YAML instead of JSON, so the client script was modified to read YAML documents instead of JSON. All in all the number of changes amounted to approximately 10 lines of code, all of which was trivial. The diff of these changes is available in Appendix B.

After the changes were made, the CI script was modified to also run the Bouncer client script and a version of the upstream API specification was placed in a directory in the SDK repository. Upon the changes being pushed to the upstream repository, the CI system picked up the Bouncer client script without issues and executed a full dependency check cycle.

However, some weeks later when the merge request containing Bouncer API checking was merged, the same dependency check executed again against the same information failed. This was found to be caused by a networking error of a particular CI runner when it was trying to contact the Bouncer instance. Further investigation indicated the error was most likely due to incorrect certificate configuration on the specific CI runner instance and not due to a fault in Bouncer.

4.2.2 Validation against a test microservice architecture

In order to validate Bouncer's ability to detect API breakages and try out the Bouncer workflow in a more dynamic situation, a sample software system utilizing the microservice architecture pattern was developed. It was decided to write a simple blogging service consisting of 5 microservices and a frontend component, which was left unimplemented, except for the dependency declarations. Since the frontend component has a dependency

on all services, it would be an easy place to detect API breakage in any service. Additionally certain services were defined as being inter-communicating by calling API endpoints in other services.

The blogging system represents a social media platform, on which users can post blog posts and follow other users whose posts they find interesting. Users can receive notifications about new posts from users they follow. Additionally they can promote their favourite profiles using a blog roll feature.

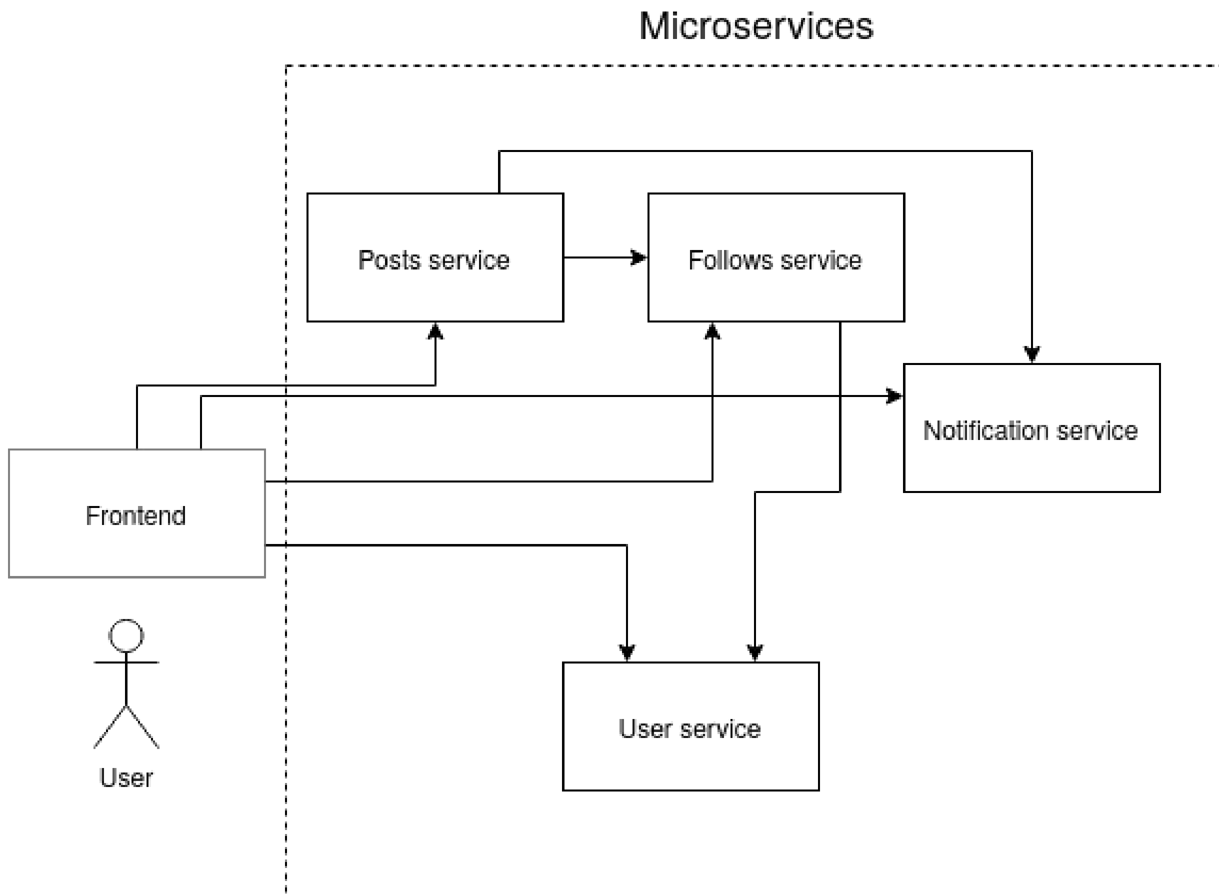

Figure 4.5: Sample microservice architecture for a blogging service

Figure 4.5 shows the service composition of the blogging system. Each of the services implements a small part of the overall system, with service responsibilities divided as so:

- User service: keeps track of user credentials and account IDs.
- Posts service: provides storage, retrieval and creation of new blog posts.
- Follows service: keeps track of which other users a particular user follows.

- Notification service: notifies a user about a new blog post.

Each service was implemented as FastAPI services in order to take full advantage of Bouncer's capabilities without need to alter the client script. Bouncer was also integrated into the CI pipeline of each service from the start in order to have all development activity covered by Bouncer API compatibility checking.

The validation case was built as a rough simulation of the development activities that would take place during a project involving the creation of microservices or APIs in general. This simulation is considered rough in the sense that all development activity was done by a single person and the development process followed didn't entirely match the structure of a realistic software project. The simulated focus was therefore mostly focused on feature creation and detecting and then subsequently fixing API breakages that naturally occured during development.

Bouncer's performance in this validation case was good, although some issues were detected during validation. These issues are covered in more detail the section on detected API breakage types and they were minor enough that they could be addressed already during validation. The issues were also concluded to have been unrelated to the principles Bouncer is based on and were either small defects in Bouncer or flaws in third-party tools. Once these issues were resolved, Bouncer reliably detected the API breakages introduced into the individual microservices and provided meaningful assistance in locating outdated API dependencies with the ability to provide a quick turnaround time from a change introduced in one service to an incompatibility detected in another one.

4.3 Adopting the system into software development process

Originally it was planned to perform a more in-depth experiment to test out how easily other developers would be able to take advantage of Bouncer by having them configure it for either a sample project or a project they were working on, if possible. However, ultimately it was not possible to organize such an experiment due to time limitations and unavailability of subjects. Therefore this section only analyzes adopting Bouncer into a software development process only from the tooling perspective. Some consideration about the human aspect of integrating Bouncer into a software development process were made, but these come from a very limited perspective.

Based on the first validation scenario, the adoption of the system to an existing development process was simple and required only approximately two man-hours for the system to be integrated into the CI pipeline of a simple SDK project. However, it should be kept in mind that the integration was done into a project that was already familiar and with perfect knowledge of the operation and setup of Bouncer. It is likely, that if the same integration was done by a person or a team without good knowledge of Bouncer, they would have taken a bit longer. On the other hand it is also worth keeping in mind that the validation case with the SDK represents an untypical use case and required extra effort that wouldn't have otherwise been necessary. Proper documentation with a step-by-step guide for integrating Bouncer into a project's CI pipeline would also help level out the differences between someone new to using Bouncer and an expert.

The integration was also possible to do incrementally with benefits gained gradually. At the very beginning of the SDK validation case, all the pieces to set up rudimentary API compatibility checks were put in place, but without taking advantage of code generation to ensure API compatibility on the code level. Generating such API bindings and taking advantage of them incrementally would be an entirely viable approach.

However, during the SDK validation case it was found that the Bouncer architecture does have potential for complications when it comes to CI system and network configurations. The Bouncer server and potential external OpenAPI document sources must be consistently accessible from the CI system. Organizations may have specific policies about which systems are accessible from where and since the Bouncer server is not secured with any kind of authentication, deploying it in a publicly accessible location is not advised.

The Bouncer client script is configured by changing the code of the script, because this was deemed to be convenient when it came to writing the Bouncer system. However, this proved to have the downside, that distinct versions of the client script end up being copied to potentially a large number of repositories. Bugs discovered in the client script require all of these scripts to be replaced and configurations rewritten into these duplicate scripts. In the future the client configuration should be a stored in a different format and the Bouncer client should be manageable using package management tools to ensure easy upgrade path for future features and bug fixes.

The overall conclusion then is that Bouncer provides an API compatibility checking system, which is easy to integrate into an existing development process and workflow within the span of a single day and applied incrementally without having to make sweeping changes to the development process. As such, Bouncer fulfills the requirement of being

integrable into CI/CD pipelines and also integrates easily into an existing software development process. During validation it was found that Bouncer does have some areas of improvement and that some complications may arise, but these issues were deemed to be either environmental, such as with the company network configuration, or minor enough that they don't require major chances to how Bouncer operates.

4.4 Which kinds of API breakages could be detected

This section covers the kinds of API breakages that were intentionally placed into the microservice scenario to test out Bouncer's ability to detect them. The details and context of each breakage is first described, followed by how Bouncer under different configurations reacted to the breakage.

Additionally, deviations from the expected behaviour are documented into the false positives and false negatives subsections to cover cases where Bouncer incorrectly identified a breakage or ignored a real breakage.

4.4.1 Altered parameters or request body

Detection of a breakage by the alteration of query parameters or request bodies was tested by modifying the Users Service in the microservice validation case by enforcing that a multi-factor authentication token was also provided to the REST endpoint responsible for logging users in. This token was first supplied as a query parameter and then as part of the request body schema.

Addition of a required query parameter was not detected as a breaking change when Bouncer was configured to use Atlassian's openapi-diff as the breakage detection tool. Exploration of the issue lead to a bug report that has been open since 2020 (*Adding query parameter is not detected as change* 2022). Testing the same OpenAPI specification files using OpenAPI Tools' openapi-diff correctly identified a breaking change, however. If Bouncer were configured to naively check for breakages the new query parameter would have also been detected at the cost of additionally false positives.

Altering the schema of a request body was correctly identified as a breaking change in all cases.

4.4.2 Altered response

Detection of an altered response schema as a breaking change was tested by modifying the User Service's login REST endpoint again such that a randomly generated token was returned in addition to a message of a successful login attempt. This token served as a representation of an authentication token, although in practice verifying the token was not implemented in any of the sample microservices. Originally the response schema was left unspecified in the FastAPI code, in which case FastAPI assumes a string value as the response.

Change to the response schema was also not detected as a breaking change when using Atlassian's openapi-diff. However, as in the case of altered parameters test case, OpenAPI Tools' openapi-diff was able to catch the breaking change correctly. Because this behavior pattern seemed to be repeating with Atlassian's openapi-diff, the decision was made to switch to the OpenAPI Tools' openapi-diff for the remainder of the testing.

4.4.3 Changed HTTP method

Detection of a changed HTTP method was tested by modifying a REST endpoint in the Follows Service which handled unfollowing a user. This endpoint was initially declared as a POST endpoint but for the purposes of testing it was changed into a DELETE endpoint.

Bouncer was able to correctly identify the breakage utilizing the OpenAPI Tools openapi-diff. The tool identified the change as the creation of a DELETE endpoint and the removal of the POST endpoint, which triggered the backwards incompatibility detection.

In a naive configuration Bouncer would have also detected the breakage, as the change is reflected clearly in the OpenAPI document.

4.4.4 Changed URL path

A changed URL path was also tested by modifying the Follows Service REST endpoint responsible for unfollowing users. This endpoint used to be named "POST /unfollow" but after changing the endpoint from a POST method to a DELETE method, it was decided to rename the endpoint to "DELETE /follow".

This change was also correctly identified as an API breakage by Bouncer while taking advantage of OpenAPI Tools openapi-diff. The tool identified that the "/unfollow" endpoint

was deleted and a new “DELETE /follow” endpoint had been created. The removal of the endpoint triggered a backwards incompatibility, as expected.

4.4.5 False positives

At the beginning of the microservice validation case the Bouncer system was set up to run in a naive way, assuming any change in the OpenAPI files was a breaking change. This was done intentionally to compare the system's performance in terms of false positives before and after adopting tools such as openapi-diff, which provide more nuance to API breakage detection.

As expected, Bouncer produced false positives in this naive operating mode, such as detecting the addition of a new endpoint to check if a user exists as a breaking change to the front-end microservice, even though in practice the existence of a new endpoint like that is unlikely to be an API breakage by itself.

After openapi-diff was integrated into the Bouncer system, which was entirely a server-side change, the same dependency check completed successfully. This provided validation that the system would not report addition of an endpoint as a false positive in the future.

Some false positives also arose when an API breakage occured in an API endpoint that was not actually in use by the dependent service. In this case the API breakage is very real, but in practice it would never have an effect on the service due to said API not being utilized in the service. Because Bouncer does not do any source code analysis, it cannot determine this difference. This meant that CI pipelines would occasionally fail in such a way that the developer would be required to update their dependency specifications in a way that could be considered unnecessary. Thanks to integration of openapi-diff, it would be possible to manually modify dependency specification such that unused API endpoints are removed. Any alteration to irrelevant API endpoints would then appear as harmless endpoint additions to openapi-diff and thus Bouncer.

4.4.6 False negatives

During the microservice architecture validation scenario a bug was found in Bouncer where a dependency check was reported as successful even though an unmatching OpenAPI document was provided as a dependency. The bug was identified when checking if a changed document would trigger a failure, which unexpectedly did not happen. This

bug was tracked down to a naming mismatch between what the dependency directory was named and what it was set to in the Bouncer configuration. The Bouncer checking routine failed to report an error for a directory that did not exists and the rest of the routine assumed no dependencies were declared and were therefore successful.

In this case the incorrect behaviour was not the result of flaws in the main part of the checking routine or the server-side component of Bouncer, but showed a flaw in the Bouncer client, which could result in confusion and service breakages going unidentified. However, since the fault was simple and identified early in the validation, it was quickly fixed in the Bouncer client and buggy versions of the client were updated.

A second false negative was the aforementioned case where Atlassian's openapi-diff did not adequately check request parameters for changes which lead to a parameter change getting past API breakage checking. This turned out to be a known defect of Atlassian's openapi-diff tool and the one from OpenAPI Tools did not exhibit the same behavior. However, this false negative does highlight the importance of properly testing and analyzing such preexisting tools when integrating them into fault detection systems like Bouncer.

4.5 Application of the system to API breakage detection

The validation cases show, that the Bouncer system can detect various kinds of API breakages and a reduction in false positives can be achieved by integrating preexisting OpenAPI specification analysis tools. Such integrations do come with some complications due to existence of similarly named tools with differing behaviour. Atlassian's openapi-diff tool was found to be generally lax with breakage detection and thus providing unreliable results, while the openapi-diff tool from OpenAPI Tools organization more consistently caught breakages. Therefore the OpenAPI document diffing tool from OpenAPI Tools should be preferred as the breakage detection tool of choice for Bouncer.

With a trustworthy OpenAPI specification diffing tool integrated, it was possible to use Bouncer to identify breakages in FastAPI services with a quick turnaround time. Such API breakages would be identified quickly in the CI system and failures reported via email in minutes. In the log output of these emails Bouncer correctly identified the API dependency which had caused the breakage and pointed the developer towards the right direction to begin solving the breakage.

The developer experience of Bouncer could still be improved, however. During this study, the Bouncer client provided a fairly minimal amount of information about the kind of breakage that had occured and was limited to just reporting a breakage of some kind had happened. The OpenAPI Tools openapi-diff tool provides very detailed information about the breakage, so this output could be sent from the Bouncer server to the client and printed out upon an API breakage.

An additional hurdle was that the Bouncer system at the moment of writing does not offer a clear workflow for updating API dependencies to address an API breakage. The Bouncer service could benefit from a web-based dashboard to view and download OpenAPI specification files that have been uploaded into the system. This way developers in the process of resolving an API breakage would have an easy way to access the up-to-date OpenAPI specification in order to begin API breakage resolution. In the microservice validation case new OpenAPI documents were downloaded by launching the services locally and then navigating to the OpenAPI document in the API documentation page generated by FastAPI, which was slow and required launching and shutting down services that were not strictly necessary for the development task at hand.

The actions a developer needs to take in order to actually fix the API breakage on the code level are out of scope for Bouncer. However, specifying a clear, recommended workflow that works well with the Bouncer system would be beneficial.

4.6 Generalizability of the system

As the SDK validation scenario shows, the Bouncer system is flexible enough that it is not limited entirely to FastAPI-based microservices, although it might not work as a drop-in solution in these cases.

The main requirement of Bouncer is that it needs some way to get up-to-date OpenAPI information from some source. In FastAPI projects that source is the FastAPI project itself and the information from other services relayed via the Bouncer service. For use with other frameworks and languages, another source would need to be found. API specifications could also always be written by hand and then submitted to Bouncer, but that risks API specifications getting out of date with the code implementing them. A realistic Bouncer-oriented workflow in an environment its designed for requires an automated way to generate accurate API specifications, which remain up to date with the service source code.

The simplicity of the system would also lend itself to be adapted for use with other communication protocols, for which there exists a standardized documentation format with similar or better level of detail as OpenAPI specification. For instance, Bouncer could compare gRPC's protobuf files instead of OpenAPI documents or API specification documents for SOAP in order to facilitate API compatibility checking for these two protocols in addition to REST. The main limitation is that to avoid false positives a similar tool to openapi-diff would need to either be found or developed to differentiate between breaking and non-breaking API changes.

So, overall the Bouncer system can be generalized across frameworks, languages and communication protocols with the following caveats:

- Bouncer must have an up-to-date source for API information (preferably automated)
- Communication protocol must have an accurate specification language, such as OpenAPI Specification
- A diffing tool with breakage detection must exist for the specification language in order to reduce false positives

Additionally some changes would need to be made to Bouncer to deal with the specification files correctly. Currently Bouncer assumes the specifications it is receiving are written in JSON, which is not necessarily the case with other API specification languages. The rest of the API breakage detection routines would essentially work as is, provided that the correct API specification diffing tool is also integrated or naive breakage detection is used.

5 Discussion

This chapter reflects on the results gained from the study and aims to put them into the correct scientific and software engineering context. To do so, the limitations of the Bouncer system are laid out and potential ways to address them are considered. Related work which was not used as background material for the study is also covered to establish a link between this study and other work aiming to address or mitigate API breakages. Finally, potential future improvements to Bouncer are considered and the study methodology is reflected upon to address study validity.

5.1 Applying the system more generally to software engineering

The Bouncer system was built with a specific development team's requirements and approaches to software development in mind. Because of this, it is worth analyzing the limitations of this system when applied to other software engineering teams and their requirements to determine the utility of the system more widely.

5.1.1 Utilizing code generation and client libraries

Bouncer can allow developers to detect API breakages on the API definition level, but it is not aware of the code calling those APIs and as such cannot detect code-level defects. It would be very much possible to declare an up-to-date API dependency but still attempt to call APIs in an incompatible way.

This issue cannot be solved with the current scope of Bouncer, but there are methods that can be used to mitigate this issue in order to maximize confidence in defect detection.

The use of code generation can assist in turning API incompatibilities into a code-level representation, therefore improving the developer's ability to make correct adjustments to account for an API change. One way to accomplish this is by using Swagger Codegen, which is a tool for the creation of server stubs and client libraries based on an OpenAPI specification (*API Code & Client Generator | Swagger Codegen* 2022).

Code generation tools would be used to create language-specific client bindings to each of the declared API dependencies based on the currently specified version of the OpenAPI specification. The software would then call these bindings rather than make requests to these APIs directly. When Bouncer detects and API breakage, the procedure to fix API compatibility would be to acquire the new OpenAPI specification for said API and then regenerate the client bindings based on the new specification.

Regular language-specific linters and syntax checkers can then be used to find areas of the program that are making calls to functions that no longer exist or make calls with missing or extra parameters. A strongly typed programming language would additionally be able to ensure parameters use matching types.

For the microservice validation case the idea of code generation was considered, but ultimately Swagger Codegen was deemed too complex for the validation case. Swagger Codegen will generate self-contained modules, which are intended to be installed separately into the development environment instead of creating individual source code files, which can be simply included in the project. Using them would have required setting up a separate package registry, which was considered too time-consuming considering the simplicity of the sample microservices.

For a serious project, code generation could still offer a good route and Swagger Codegen could be integrated into the CI/CD pipeline to automatically create and upload API bindings to an internal package registry for other services to consume.

5.1.2 Direct microservice communication versus event-based communication

Direct communication between microservices has been determined to be complex and as a pattern is discouraged in favour of either scoping each microservice such that it can solve each use-case in isolation or communicate with other microservices through event-based communication methods (Cerny et al., 2018). Event-based communication aids in service decoupling and as long as the format of events is commonly shared, microservices need not know of each other and the communication protocol problem is also removed, since all communication is now based on a single publisher-subscriber communication channel.

This means that the findings in this thesis only apply to a specific type of microservice architectures, in which services are communicating directly. As already described in this thesis, this direct communication comes with many challenges. However, (Zhang et al.,

2019) shows that REST-based directly communicating microservices are being made by industry practitioners.

With the artifact developed in this thesis, the direct-communication pattern may also become more approachable as some of the challenges are mitigated by improved testing procedure. The direct-communication pattern also eliminates the need for a shared communication channel, which may simplify the architecture as a whole since communication between microservices is analogous to the communication between a client and the microservice architecture. Application of the techniques described in this thesis may then lead to the wider adoption of direct-communicating microservices as a pattern.

Another potential avenue for use of the artifact and techniques is in architectures based on functions-as-a-service (FaaS). Since these functions are short-lived, they cannot utilize publisher-subscriber techniques as well as long-running services, which leaves their public interfaces as the only realistic method for cross-function communication. These FaaS interfaces would then be similar to direct-communication interfaces between microservices and they can be mapped into a similar SDG of providers and dependencies.

5.1.3 Organization-level adoption

Bouncer in its current state was built in the context of a team that focuses on rapid prototyping and was designed to meet the requirements of that kind of an environment. Many of these requirements don't apply to various other contexts, such as in production development work. Such teams should have the time and resources available to conduct comprehensive quality assurance and testing and the bar for an API change is likely significantly higher than in a prototype project.

Bouncer also has certain technical limitations that restrict how it can be used inside an organization. A Bouncer instance stores its database as an SQLite database file locally and since the state of API stability is only considered on a per instance basis, this limits horizontal scaling. Additionally Bouncer considers services inside a single unified namespace of services. Multiple versions of the same service would therefore need to be explicitly named differently in order to be submitted into the same Bouncer server instance. As a result, Bouncer is currently developed to operate at the scale of single development teams operating on a single project per Bouncer instance.

Organizations also may deal with multiple types of APIs and not just REST. Such generalizability was covered in chapter 4 and Bouncer should be a simple and flexible enough

system to be adapted for this kind of use. However, it is not there yet and would require additional work to make Bouncer work fully with multiple API types.

Where Bouncer may find use are in teams producing quick prototypes, which operate in a similar way to the Advanced Technology Group. Additionally parts of Bouncer's design or even Bouncer as a whole could be used as a safety check to prevent unexpected API breakages even if the teams developing those APIs are not necessarily taking advantage of Bouncer as a quick API breakage detection tool as part of their development workflow. This would most likely be accomplished by relaying mostly on openapi-diff and similar tools combined possibly with a small amount of code from the Bouncer project.

5.2 Related work

This section covers scientific work and technologies which are relevant to API breakage detection or mitigation techniques and contrasts them to the ones used in Bouncer.

5.2.1 Contract-first development

Bouncer was created for a development workflow based on a code-first approach, meaning that API specifications are derived from the implementation and is altered to match changes in the implementation. The use of FastAPI in projects at Advanced Technology Group was specifically driven by its ability to generate documentation from code level constructs, since the code is expected to change even radically during prototyping.

An alternative approach to API specifications exists, where the API specification is created first and implementations carefully follow the specification rather than the other way around (Zhong and Yang, 2009). This approach is referred to as *c*ontract-first design, because of the specification's role as a binding contract between the API producer and consumer.

In a contract-first approach the process of changing an API is different, because the specification is held at a higher standard. Any changes would therefore necessarily have to be represented in the specification first and the correct procedures for altering the specification would need to be followed. If the consuming parties are appropriately involved in these procedures, they would also learn about any changes in advance.

A contract-first approach with a mandatory and clearly defined procedure for making

changes would result in accidental breakages being in the specification having a higher chance of being detected early and consuming parties being present in the discussions would also help to inform API consumers early of necessary compatibility breakages.

5.2.2 Other inter-service interface definition languages

REST is naturally not the only communication protocol for inter-service communication, nor is OpenAPI the only way to describe API endpoints. So, while Bouncer was designed purely with REST and OpenAPI in mind, other API protocols and their interface definition languages (IDL) are worth considering.

A predecessor to REST was the SOAP protocol, which also has its own IDL in the form of WSDL (Curbera et al., 2002). Similar to OpenAPI, it allows developers to specify the input and output values of API endpoints in enough detail for code generation.

Another option is the Google-developed gRPC protocol (*gRPC* 2022). gRPC solves many of the research problems of this thesis by requiring, by default, a strict declaration of communication endpoints and data using an interface description language called Protocol Buffers (Protobuf). These protobufs are used to describe the communication protocol and code generation is then used to make the correct calls to the service in question.

Amazon has also created its own IDL called Smithy (*Smithy — Smithy 1.0 documentation* 2022), which is intended to be protocol-agnostic and allow different types of APIs to be described with the same language. Smithy puts similar emphasis on code generation to OpenAPI and a diffing tool similar to openapi-diff also exists for Smithy in the form of Smithy Diff.

While the current version of Bouncer was built to detect API breakages in REST APIs using OpenAPI specifications, it could be adapted to function with other IDLs like WSDL, Protobuf or Smithy. At its most basic level, Bouncer is only interested in whether two supplied API specifications differ from one-another, which could be easily accomplished in an entirely IDL-agnostic way. In practice, however, Bouncer is more useful when paired with a proper tool for detecting breaking changes. For Smithy this would be trivial as such a tool already exists. For other IDLs similar tools should either be found or developed first, but when such tools are available they could be integrated quickly into Bouncer.

5.2.3 Comprehensive automatic test generation

Bouncer approaches the problem of API compatibility testing from the point of view of static analysis in order to minimize the amount of effort needed to gain some level of trust in the state of inter-service communication. Use of techniques inspired by static analysis makes the process fairly light-weight, although it does suffer from the possibility of false positives and requires a level of discipline from the developer team using a Bouncer-oriented workflow to not deviate from API communication, which Bouncer can analyze.

Similar benefits could be gained by approaching the problem from software testing side by reducing the burden of test creation by creating tests automatically. One approach for generating tests would be to use traffic monitoring during the use of the system to synthesize realistic test cases by simply interacting with the software system (Gazzola et al., 2020). Some level of manual testing will need to be performed on the software system anyway, and this time could be used to derive test cases at the same time.

It would also be possible to use the same OpenAPI documents that Bouncer uses to instead automatically generate API tests, through which breaking API changes could be detected (Ed-douibi et al., 2018). This approach does not necessarily test inter-service communication, but it ensures APIs do not change unexpectedly and a system of communicating these changes could be built on top of the work of detecting a breaking API change.

5.2.4 Language-based approach to microservice development

In this study, testing the API compatibility of inter-service communication has occasionally been compared to the concept of syntax checking in programming languages. The idea being that network API calls have similarities to function calls in programming languages, but function calls can be syntax and type checked for correctness. If inter-service communication were represented in the format of a programming language, the process of checking inter-service communication correctness would just be a matter of syntax and type checking.

This kind of a language-based approach has seen some study (Guidi et al., 2017), and languages are available that allow service endpoints to be declared as source code with type information, such as the Jolie language. This approach makes it possible for service interactions to be strongly type-checked to ensure compatibility.

However, this approach does also involve trading off some of the benefits of the microservice approach. The teams would all need to use a common programming language to implement their API endpoint exposure, so that the composition of services could be checked properly and the integration would likely need to happen in a shared source code repository to make sure interface declarations are always up-to-date between all of the services. So, this approach is of limited utility in situations, where large numbers of teams develop separate APIs. However, for a group developing microservices for a specific project with a willingness to use a shared source code repository and programming language it may provide increased benefits over the Bouncer approach.

5.3 Future work

The purpose of this study was to discover a technique for detecting API breakages in a rapid prototyping context in order to speed up debugging inter-service communication issues and focus on building prototypes rather than test suites. A technique to accomplish this was found and implemented in the form of Bouncer and its suitability for API breakage detection was validated with success using two validation cases.

The next step in the development of Bouncer is to improve its usability and to explore its value in an organizational context and potential plans for adoption by development teams. Initially there were some plans to involve trainees in the validation testing of Bouncer, but this didn't come to pass due to time constraints. Therefore in the future it would be valuable to investigate how an entire team could take advantage of Bouncer.

Studying other API specification languages and integrating them into Bouncer would also offer an interesting avenue for more research and development. While this has been theorized to be easy to accomplish, it should also be tried in practice to confirm the hypothesis.

While some recommendations about the workflows that can be implemented with Bouncer were covered in this study, they were only touched upon at a very general level and without properly trying them out. Testing out approaches that utilize code generation for generating client code from API specifications in order to make solving API breakages a clearer task for a programmer should be explored and ways to extend Bouncer to accommodate such workflows considered.

5.4 Threats to validity

This study is based on the design science research methodology, which means that the aim of the study was to provide a solution to a predetermined problem, which in the case of this study was detecting API breakages in REST APIs with minimal need for the developer to write test cases themselves. As can be seen from the problem statement, this problem exists in a particular context and thus solutions to it are also dependent on the context. Therefore general conclusions about how API breakages can or should be detected cannot be drawn alone from this case. The results from this study offer a specific perspective, which could be used to inform a general consensus if analyzed alongside other studies using a rigorous methodology, such as a systematic literature review.

The study was also conducted largely independently and the system design requirements and validation cases were determined with fairly little direct oversight from potential stakeholders. Therefore the validation cases are unlikely to be comprehensive enough to showcase quality required for full adoption on an organizational level. While some potential additional requirements for the Bouncer system's organizational adoption were considered, these remain hypothetical. The general requirements for the system and how the system might function were established in collaboration with peer stakeholders and presented to key members of the Advanced Technology Group team, so main stakeholders did offer at least a general approval of the system's design and requirements.

Additionally the results of this study are generally qualitative rather than quantitative. Claims about the difficulty of a task are inherently subjective and since this study was performed without separate study subjects, they were only possible to make from the perspective of the artifact designer. This may result in bias, since the claims are made by someone with intimate understanding of the system and potential conflicting interests to present it in a good light.

In order to try and minimize the impact of these threats, certain steps were taken during the study. Firstly, in parts where qualitative claims were made, there was an attempt to be transparent about biases and assumptions that may have affected that conclusion. Secondly, such claims were backed up with the metrics that were available in each case, such as amount of time spent or the amount of changes required.

However, the most important step to address validity in this study was by being as transparent about the scientific method used, activities that were conducted at different phases of the study and the design of the Bouncer system. The specification of the design artifact

was described as clearly as possible and various validation scenarios and the steps taken within them were described in detail. All of this was also supplemented with architectural and sequence diagrams and code snippets and results from code coverage tools were included as appendix pages for further study. This rich description was aimed to relay as much context as possible to the reader, so that the reader may draw their evaluate the claims with an adequate amount of information. The objective was to be so clear about the system's functionality and design that a reader with moderate programming skills could reproduce the system just from the description alone.

6 Conclusions

The product of this study was an automated system that could be combined with the development methodology used at Nokia Advanced Technology Group to create REST-based web services with automatic API breakage detection between inter-communicating web services. This automated system, called Bouncer, stores OpenAPI documents of tracked services and executes API breakage detection by comparing an up-to-date document it keeps in a database with those submitted by dependent services. During validation two approaches were tested for the API breakage detection, one being a naive approach of assuming all changes were breaking and another approach that cut down on false positives by utilizing pre-made OpenAPI diffing tools.

During validation the Bouncer system proved flexible and integrating it into an existing software development process required a small amount of work, approximately on an hour for setting up an instance of the Bouncer service and configuring the continuous integration pipeline to execute a Bouncer API compatibility check between an internal SDK and its related API specification. While some difficulties with the CI integration were noted, these were caused by issues with SSL certificates and company firewall rules and as such were not flaws of the Bouncer system.

Bouncer's API breakage detection was tested in a validation case, where a full microservice architecture of inter-communicating REST services was created and various breakages were introduced to different services and Bouncer's reaction to these breakages was recorded. After some initial difficulties with Atlassian's openapi-diff, eventually a usable tool for comparing OpenAPI documents was successfully integrated and all categories of API breakages documented in Chapter 3 were successfully caught by Bouncer while also reducing the false positives in cases where a change did not affect existing API consumers.

The Bouncer system's merits for wider adoption and flexibility in terms of potential use of different interface definition languages was also considered and Bouncer was concluded to be simple and adaptable enough that it could be used with other types of APIs beyond REST with some modifications. However, Bouncer provides the most utility for lightweight API breakage detection when combined with a way to generate API specifications from source code in a way similar to how FastAPI produces OpenAPI documents.

While Bouncer does have limitations and is quite heavily reliant on auxiliary utilities

such as openapi-diff, the approach it demonstrates seems viable for assisting developers with the detection and fixing of API breakages. A number of avenues to improve Bouncer further were also identified and by pursuing them, a comprehensive system for light-weight API breakage detection could be developed with improved developer comfort and greater coverage of different API types and frameworks.

Bibliography

Adding query parameter is not detected as change (2022). en. URL: https://bitbucket.org/atlassian/openapi-diff/issues/21/adding-query-parameter-is-not-detected-as (visited on 07/12/2022).

Albano, M. and Nielsen, B. (June 2020). "Interoperability by construction: code generation for Arrowhead Clients". In: *2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*. Vol. 1, pp. 429–432. DOI: 10.1109/ICPS48405.2020.9274746.

API Code & Client Generator | Swagger Codegen (2022). URL: https://swagger.io/tools/swagger-codegen/ (visited on 06/07/2022).

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., and Penix, J. (Sept. 2008). "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5. Conference Name: IEEE Software, pp. 22–29. ISSN: 1937-4194. DOI: 10.1109/MS.2008.130.

Budde, R. and Zullighoven, H. (May 1990). "Prototyping revisited". In: *COMPEURO'90: Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering - Systems Engineering Aspects of Complex Computerized Systems*, pp. 418–427. DOI: 10.1109/CMPEUR.1990.113653.

Cerny, T., Donahoo, M. J., and Trnka, M. (Jan. 2018). "Contextual understanding of microservice architecture: current and future directions". In: *ACM SIGAPP Applied Computing Review* 17.4, pp. 29–45. ISSN: 1559-6915. DOI: 10.1145/3183628.3183631. URL: https://doi.org/10.1145/3183628.3183631 (visited on 03/08/2022).

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., and Weerawarana, S. (Mar. 2002). "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI". In: *IEEE Internet Computing* 6.2, pp. 86–93. ISSN: 1089-7801. DOI: 10.1109/4236.991449. URL: http://ieeexplore.ieee.org/document/991449/ (visited on 08/01/2022).

Ed-douibi, H., Cánovas Izquierdo, J. L., and Cabot, J. (Oct. 2018). "Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach". In: *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*. ISSN: 2325-6362, pp. 181–190. DOI: 10.1109/EDOC.2018.00031.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). "Microservices: Yesterday, Today, and Tomorrow". en. In: *Present and Ulterior Software Engineering*. Ed. by M. Mazzara and B. Meyer. Cham: Springer International Publishing, pp. 195–216. ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-

319-67425-4_12. URL: https://doi.org/10.1007/978-3-319-67425-4_12 (visited on 03/08/2022).

Espinha, T., Zaidman, A., and Gross, H.-G. (Feb. 2015). "Web API growing pains: Loosely coupled yet strongly tied". en. In: *Journal of Systems and Software* 100, pp. 27–43. ISSN: 0164-1212. DOI: 10.1016/j.jss.2014.10.014. URL: https://www.sciencedirect.com/science/article/pii/S0164121214002180 (visited on 05/12/2022).

FastAPI (2022). URL: https://fastapi.tiangolo.com/ (visited on 04/26/2022).

Feng, X., Shen, J., and Fan, Y. (Oct. 2009). "REST: An alternative to RPC for Web services architecture". In: *2009 First International Conference on Future Information Networks*, pp. 7–10. DOI: 10.1109/ICFIN.2009.5339611.

Gazzola, L., Goldstein, M., Mariani, L., Segall, I., and Ussi, L. (Oct. 2020). "Automatic Ex-Vivo Regression Testing of Microservices". In: *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. AST '20. New York, NY, USA: Association for Computing Machinery, pp. 11–20. ISBN: 978-1-4503-7957-1. DOI: 10.1145/3387903.3389309. URL: https://doi.org/10.1145/3387903.3389309 (visited on 02/14/2022).

GitHub - OpenAPITools/openapi-diff (2022). *GitHub - OpenAPITools/openapi-diff: Utility for comparing two OpenAPI specifications.* en. URL: https://github.com/OpenAPITools/openapi-diff (visited on 07/12/2022).

gRPC (2022). en. URL: https://grpc.io/ (visited on 04/14/2022).

Guidi, C., Lanese, I., Mazzara, M., and Montesi, F. (2017). "Microservices: A Language-Based Approach". en. In: *Present and Ulterior Software Engineering*. Ed. by M. Mazzara and B. Meyer. Cham: Springer International Publishing, pp. 217–225. ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_13. URL: https://doi.org/10.1007/978-3-319-67425-4_13 (visited on 05/12/2022).

Janzen, D. and Saiedian, H. (Sept. 2005). "Test-driven development concepts, taxonomy, and future direction". In: *Computer* 38.9. Conference Name: Computer, pp. 43–50. ISSN: 1558-0814. DOI: 10.1109/MC.2005.314.

Jovanovic, N., Kruegel, C., and Kirda, E. (May 2006). "Pixy: a static analysis tool for detecting Web application vulnerabilities". In: *2006 IEEE Symposium on Security and Privacy (S P'06)*. ISSN: 2375-1207, 6 pp.–263. DOI: 10.1109/SP.2006.29.

Karavisileiou, A., Mainas, N., and Petrakis, E. G. (Nov. 2020). "Ontology for OpenAPI REST Services Descriptions". In: *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*. ISSN: 2375-0197, pp. 35–40. DOI: 10.1109/ICTAI50040.2020.00016.

Karlsson, S., Čaušević, A., and Sundmark, D. (Oct. 2020). "QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. ISSN: 2159-4848, pp. 131–141. DOI: 10.1109/ICST46399.2020.00023.

Ma, S.-P., Fan, C.-Y., Chuang, Y., Liu, I.-H., and Lan, C.-W. (Nov. 2019). "Graph-based and scenario-driven microservice analysis, retrieval, and testing". en. In: *Future Generation Computer Systems* 100, pp. 724–735. ISSN: 0167-739X. DOI: 10.1016/j.future.2019.05.048. URL: https://www.sciencedirect.com/science/article/pii/S0167739X19302614 (visited on 02/14/2022).

Machado, P., Vincenzi, A., and Maldonado, J. C. (2010). "Software Testing: An Overview". en. In: *Testing Techniques in Software Engineering: Second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures*. Ed. by P. Borba, A. Cavalcanti, A. Sampaio, and J. Woodcook. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 1–17. ISBN: 978-3-642-14335-9. DOI: 10.1007/978-3-642-14335-9_1. URL: https://doi.org/10.1007/978-3-642-14335-9_1 (visited on 02/28/2022).

Meyer, M. (May 2014). "Continuous Integration and Its Tools". In: *IEEE Software* 31.3. Conference Name: IEEE Software, pp. 14–16. ISSN: 1937-4194. DOI: 10.1109/MS.2014.58.

OpenAPI Initiative (2022). en-US. URL: https://www.openapis.org/ (visited on 03/28/2022).

openapi-diff (2022). en. URL: https://www.npmjs.com/package/openapi-diff (visited on 07/12/2022).

Pacheco, V. F. (2018). *Microservice Patterns and Best Practices: Explore Patterns Like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices*. Birmingham, UNITED KINGDOM: Packt Publishing, Limited. ISBN: 978-1-78847-120-6. URL: https://ebookcentral-proquest-com.libproxy.helsinki.fi/lib/helsinki-ebooks/detail.action?pq-origsite=primo&docID=5259455 (visited on 02/15/2022).

Pearce, D. J. (Apr. 2021). "A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust". In: *ACM Transactions on Programming Languages and Systems* 43.1, 3:1–3:73. ISSN: 0164-0925. DOI: 10.1145/3443420. URL: https://doi.org/10.1145/3443420 (visited on 05/03/2022).

Shahin, M., Ali Babar, M., and Zhu, L. (2017). "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices".

In: *IEEE Access* 5. Conference Name: IEEE Access, pp. 3909–3943. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2685629.

Smithy — Smithy 1.0 documentation (2022). URL: https://awslabs.github.io/smithy/ (visited on 08/01/2022).

Spadini, D., Aniche, M., Bruntink, M., and Bacchelli, A. (May 2017). "To Mock or Not to Mock? An Empirical Study on Mocking Practices". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. Buenos Aires, Argentina: IEEE, pp. 402–412. ISBN: 978-1-5386-1544-7. DOI: 10.1109/MSR.2017.61. URL: http://ieeexplore.ieee.org/document/7962389/ (visited on 07/25/2022).

Vincenzi, A., Delamaro, M., Höhn, E., and Maldonado, J. C. (2010). "Functional, Control and Data Flow, and Mutation Testing: Theory and Practice". en. In: *Testing Techniques in Software Engineering: Second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures*. Ed. by P. Borba, A. Cavalcanti, A. Sampaio, and J. Woodcook. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 18–58. ISBN: 978-3-642-14335-9. DOI: 10.1007/978-3-642-14335-9_2. URL: https://doi.org/10.1007/978-3-642-14335-9_2 (visited on 02/28/2022).

Wieringa, R. (May 2010). "Design science methodology: principles and practice". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ICSE '10. New York, NY, USA: Association for Computing Machinery, pp. 493–494. ISBN: 978-1-60558-719-6. DOI: 10.1145/1810295.1810446. URL: https://doi.org/10.1145/1810295.1810446 (visited on 02/24/2022).

Zhang, H., Li, S., Jia, Z., Zhong, C., and Zhang, C. (Mar. 2019). "Microservice Architecture in Reality: An Industrial Inquiry". In: *2019 IEEE International Conference on Software Architecture (ICSA)*, pp. 51–60. DOI: 10.1109/ICSA.2019.00014.

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J., and Vouk, M. (Apr. 2006). "On the value of static analysis for fault detection in software". In: *IEEE Transactions on Software Engineering* 32.4. Conference Name: IEEE Transactions on Software Engineering, pp. 240–253. ISSN: 1939-3520. DOI: 10.1109/TSE.2006.38.

Zhong, Y. and Yang, J. (July 2009). "Contract-First Design Techniques for Building Enterprise Web Services". In: *2009 IEEE International Conference on Web Services*, pp. 591–598. DOI: 10.1109/ICWS.2009.91.

Appendix A Example OpenAPI specification

The following is an automatically generated OpenAPI specification for the User Service written as part of the microservice validation case.


```
{
  "openapi": "3.0.2",
  "info": {
    "title": "user-service",
    "description": "Service for user management",
    "version": "0.1"
  },
  "paths": {
    "/": {
      "get": {
        "summary": "Read Root",
        "operationId": "read_root__get",
        "responses": {
          "200": {
            "description": "Successful Response",
            "content": {
              "application/json": {
                "schema": {}
              }
            }
          }
        }
      }
    },
    "/register": {
      "post": {
        "summary": "Register User",
        "operationId": "register_user_register_post",
        "requestBody": {
          "content": {
```

```
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/User"
              }
            }
          },
          "required": true
        },
        "responses": {
          "200": {
            "description": "Successful Response",
            "content": {
              "application/json": {
                "schema": {}
              }
            }
          },
          "422": {
            "description": "Validation Error",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/HTTPValidationError"
                }
              }
            }
          }
        }
      }
    },
    "/login": {
      "post": {
        "summary": "Login User",
        "operationId": "login_user_login_post",
        "requestBody": {
```

```
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/LoginInfo"
            }
          }
        },
        "required": true
      },
      "responses": {
        "200": {
          "description": "Successful Response",
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/LoginResponse"
              }
            }
          }
        },
        "422": {
          "description": "Validation Error",
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/HTTPValidationError"
              }
            }
          }
        }
      }
    }
  },
  "/exists/{username}": {
    "get": {
```

```
      "summary": "User Exists",
      "operationId": "user_exists_exists__username__get",
      "parameters": [
        {
          "required": true,
          "schema": {
            "title": "Username",
            "type": "string"
          },
          "name": "username",
          "in": "path"
        }
      ],
      "responses": {
        "200": {
          "description": "Successful Response",
          "content": {
            "application/json": {
              "schema": {}
            }
          }
        },
        "422": {
          "description": "Validation Error",
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/HTTPValidationError"
              }
            }
          }
        }
      }
    }
  }
```

```
  },
  "components": {
    "schemas": {
      "HTTPValidationError": {
        "title": "HTTPValidationError",
        "type": "object",
        "properties": {
          "detail": {
            "title": "Detail",
            "type": "array",
            "items": {
              "$ref": "#/components/schemas/ValidationError"
            }
          }
        }
      },
      "LoginInfo": {
        "title": "LoginInfo",
        "required": [
          "name",
          "password",
          "mfa"
        ],
        "type": "object",
        "properties": {
          "name": {
            "title": "Name",
            "type": "string"
          },
          "password": {
            "title": "Password",
            "type": "string"
          },
          "mfa": {
            "title": "Mfa",
```

```
        "type": "string"
      }
    }
  },
  "LoginResponse": {
    "title": "LoginResponse",
    "required": [
      "status",
      "token"
    ],
    "type": "object",
    "properties": {
      "status": {
        "title": "Status",
        "type": "string"
      },
      "token": {
        "title": "Token",
        "type": "string"
      }
    }
  },
  "User": {
    "title": "User",
    "required": [
      "name",
      "password"
    ],
    "type": "object",
    "properties": {
      "name": {
        "title": "Name",
        "type": "string"
      },
      "password": {
```

```
          "title": "Password",
          "type": "string"
        }
      }
    },
    "ValidationError": {
      "title": "ValidationError",
      "required": [
        "loc",
        "msg",
        "type"
      ],
      "type": "object",
      "properties": {
        "loc": {
          "title": "Location",
          "type": "array",
          "items": {
            "anyOf": [
              {
                "type": "string"
              },
              {
                "type": "integer"
              }
            ]
          }
        },
        "msg": {
          "title": "Message",
          "type": "string"
        },
        "type": {
          "title": "Error Type",
          "type": "string"
```

```
                    }
                }
            }
        }
    }
}
```

Appendix B Bouncer Client source code

```
#!/usr/bin/env python3

# Change this import to point to the FastAPI "app" for the current project
from bouncer import app

SERVICE_NAME = "<replace-me>"
BOUNCER_INSTANCE = "http://localhost:8000"
DEPENDENCY_DIR = "./tests/test_dependencies/"

import requests
import os, sys
import json

def get_provider_openapi():
    openapi_json = app.openapi()

    return openapi_json

def upload_provider_openapi_to_bouncer(openapi_object):
    SERVICE_NAME = openapi_object["info"]["title"]

    payload = {
        "service_name": SERVICE_NAME,
        "openapi": openapi_object
    }

    res = requests.post(f"{BOUNCER_INSTANCE}/provider", json=payload)

    if not res.ok:
        raise ProviderUploadError(SERVICE_NAME, res.reason, res.text)
```

```
def get_dependencies():
    collected_files = []

    if not os.path.isdir(DEPENDENCY_DIR):
        raise DependencyCheckError(SERVICE_NAME, "DEPENDENCY_DIR does not exi

    for (root, dirs, files) in os.walk(DEPENDENCY_DIR):
        for file in files:
            collected_files.append(os.path.join(root, file))

    return collected_files

def upload_dependency_openapi_to_bouncer(filename):
    with open(filename) as f:
        text = f.read()

        openapi_object = json.loads(text)
        DEPENDENCY_NAME = openapi_object["info"]["title"]

        payload = {
            "service_name": SERVICE_NAME,
            "dependency_name": DEPENDENCY_NAME,
            "openapi": openapi_object
        }

        res = requests.post(f"{BOUNCER_INSTANCE}/dependency", json=payload)

        if not res.ok:
            raise DependencyCheckError(SERVICE_NAME, res.reason, res.text)


class ProviderUploadError(Exception):
    """Represent a failed dependency check"""

class DependencyCheckError(Exception):
```

```
    """Represent a failed dependency check"""

def main():
    print("Uploading endpoint information...")
    try:
        upload_provider_openapi_to_bouncer(get_provider_openapi())
    except ProviderUploadError as err:
        print(f"FAILED: {err}")
        sys.exit(os.EX_SOFTWARE)

    print("Finding dependencies...")
    dependencies = get_dependencies()

    print("Uploading dependencies...")
    for dep in dependencies:
        print(f"Dependency: {dep}")
        try:
            upload_dependency_openapi_to_bouncer(dep)
        except DependencyCheckError as err:
            print(f"FAILED: {dep} | {err}")
            sys.exit(os.EX_SOFTWARE)

    print("All good!")

if __name__ == "__main__":
    main()
```

Appendix C Bouncer Client diff listing

```
#!/usr/bin/env python3

-# Change this import to point to the FastAPI "app" for the current project
-from bouncer import app
-
-SERVICE_NAME = "<replace-me>"
-BOUNCER_INSTANCE = "http://localhost:8000"
-DEPENDENCY_DIR = "./tests/test_dependencies/"
+SERVICE_NAME = "<PROJECT NAME REMOVED>"
+BOUNCER_INSTANCE = "<VM URL REMOVED>"
+DEPENDENCY_DIR = "./api-tests/api-dependencies/"

 import requests
 import os, sys
 import json
+import yaml

 def get_provider_openapi():
-    openapi_json = app.openapi()
+    print("Fetching API document...")
+    res = requests.get("<GIT REPOSITORY URL REMOVED>", verify = False)
+
+    if not res.ok:
+        raise ProviderUploadError(res.reason)

-    return openapi_json
+    openapi_object = yaml.safe_load(res.text)
+
+    print("Fetch successful.")
+
+    return openapi_object
```

```
 def upload_provider_openapi_to_bouncer(openapi_object):
     SERVICE_NAME = openapi_object["info"]["title"]
@@ -42,7 +48,7 @@ def upload_dependency_openapi_to_bouncer(filename):
     with open(filename) as f:
         text = f.read()

-        openapi_object = json.loads(text)
+        openapi_object = yaml.safe_load(text)
         DEPENDENCY_NAME = openapi_object["info"]["title"]

         payload = {
@@ -65,10 +71,12 @@ class DependencyCheckError(Exception):

 def main():
     print("Uploading endpoint information...")
+
     try:
         upload_provider_openapi_to_bouncer(get_provider_openapi())
     except ProviderUploadError as err:
         print(f"FAILED: {err}")
+        sys.exit(os.EX_SOFTWARE)

     print("Finding dependencies...")
     dependencies = get_dependencies()
```

Appendix D Bouncer test coverage

```
------------ coverage: platform linux, python 3.9.9-final-0 ------------
Name                          Stmts   Miss  Cover
-------------------------------------------------
bouncer/__init__.py               2      0   100%
bouncer/crud.py                  27      0   100%
bouncer/database.py               7      0   100%
bouncer/main.py                  78      5    94%
bouncer/models.py                13      0   100%
bouncer/openapi_parser.py        48      3    94%
bouncer/schemas.py               23      0   100%
bouncer/sdg.py                   57      0   100%
-------------------------------------------------
TOTAL                           255      8    97%
```