AutoGuard: Reporting Breaking Changes of REST APIs from Java Spring Boot Source Code

Alexander Lercher, Clemens Bauer, Christian Macho, Martin Pinzger
Department of Informatics Systems
University of Klagenfurt
Klagenfurt, Austria
{alexander.lercher, clemens.bauer, christian.macho, martin.pinzger}@aau.at

***Abstract*—REpresentational State Transfer (REST) Application Programming Interfaces (APIs) are widely used for the communication between loosely coupled web services. While the loose coupling allows services to evolve independently, it requires development teams to actively identify changes in their REST APIs to notify teams of affected services. If overlooked, changes can result in unexpected breaking changes that lead to failures in the affected service.**

In this paper, we present AutoGuard, our tool for automatically extracting and reporting breaking changes of REST APIs in services developed with the popular Java Spring Boot framework. AutoGuard consists of two components: the first component generates the OpenAPI descriptions of two versions of a REST API from the source code; the second component extracts and logs the differences between them and reports the API breaking changes. AutoGuard's static analysis does not require running a service to understand its REST API changes, enabling direct integration into the development process.

We integrated AutoGuard into GitHub's and GitLab's continuous integration workflows, where it automatically generates the REST API change logs for pull and merge requests and reports any breaking changes. With this information, developers and code reviewers can make informed decisions on how to proceed with the requests.

Video demonstration: https://youtu.be/3qeWIVfMvWE

Tool repository: https://github.com/MSA-API-Management/AutoGuard

***Index Terms*—API Evolution, REST API, Backward compatibility, Continuous integration**

I. Introduction

REpresentational State Transfer (REST) [1] is widely used to expose Application Programming Interfaces (APIs) of web services and facilitates the development of loosely coupled systems. Each web service provides specific functionality and communicates with other services exclusively over public REST APIs [2]. This setup promotes maintainability, scalability, and fault-tolerance and allows services to evolve independently [3], [4].

As each service's REST API evolves independently, maintaining compatibility across services requires more coordination between development teams than in a monolithic architecture [5], [6]. Changes in one REST API could result in unexpected behavior or break the functionality of dependent consumer services. From the consumer perspective, breaking changes are undetected at build time and only manifest when the changed API is called at runtime. Consumer-side contract tests do not mitigate this problem, as API test doubles do not automatically represent the latest version. Accordingly, developers rely on other teams to correctly and timely notify them of any changes in the API.

As found in our previous study [7], REST API developers encounter difficulties in understanding the impact of source code changes on the APIs of their web services. This problem is exacerbated with larger and more complex systems. In the worst case, API developers miss a breaking change, resulting in unexpected failures on the consumer side during operation. Accordingly, previous studies have proposed detecting incompatible API changes as early as possible in the software development process as an important open research direction [7], [8].

Addressing this direction, we present *AutoGuard*, our tool for extracting and reporting breaking REST API changes in web services developed with the popular Java framework Spring Boot [9]. It guards such services against source code changes that introduce unexpected breaking API changes. AutoGuard uses AutoOAS [10], our novel approach for generating accurate OpenAPI descriptions of REST APIs directly from source code. Furthermore, it integrates OpenAPI-diff [11] for identifying differences between two versions of an OpenAPI description. Based on the differences, it reports and highlights breaking REST API changes. We integrated AutoGuard into GitHub's and GitLab's continuous integration (CI) workflows to facilitate industry adoption.

The remainder of this paper is structured as follows. Section II presents the architecture and inner workings of AutoGuard. Section III presents a typical use case of AutoGuard with an example project. Related work is presented in Section IV and Section V concludes the paper.

II. AutoGuard tool

AutoGuard consists of two components that are used in the workflow shown in Figure 1. First, *AutoGuard-gen* is executed twice to generate OpenAPI descriptions for two versions of a REST API developed with Java Spring Boot. Second, *AutoGuard-diff* compares the two OpenAPI descriptions, reports all API changes, and fails the CI pipeline on breaking changes. This information helps developers and code

This research was funded in whole or in part by the Austrian Science Fund (FWF) 10.55776/P36698.

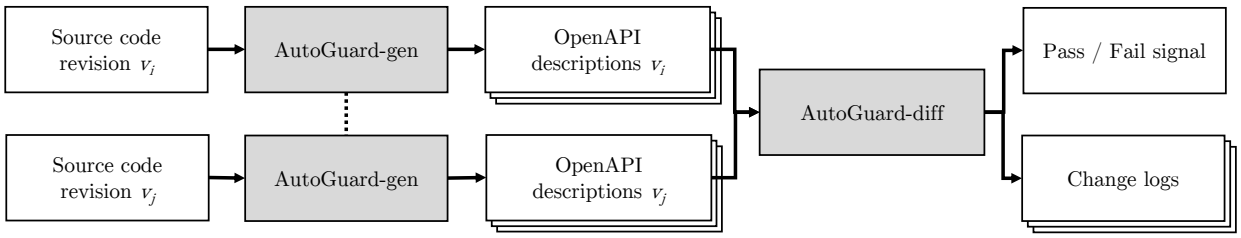

Fig. 1. Overview of AutoGuard's workflow, consisting of two components (gray) and their input and output artifacts (white).

TABLE I
SPRING BOOT ANNOTATIONS CONSIDERED BY AUTOOAS FOR GENERATING OPENAPI DESCRIPTIONS

Analysis Step	Spring Boot Annotations
Controller classes	@Controller, @RestController, @RepositoryRestController
Spring profiles	@Profile
Controller-advice classes	@ControllerAdvice, @RestControllerAdvice
REST method definitions	@PostMapping, @PutMapping, @PatchMapping, @GetMapping, @DeleteMapping, @RequestMapping
Parameter definitions	@PathVariable, @RequestParam, @RequestHeader, @RequestBody, @ModelAttribute
Response status codes	@ResponseStatus
Exception handler methods	@ExceptionHandler
Required data model fields	@NotNull, @NotEmpty

reviewers understand the impact of source code changes on a web service's REST API.

We integrated AutoGuard into GitHub and GitLab to facilitate industry adoption. AutoGuard and its source code are available on GitHub[1].

In the following, we describe the two components in detail. Note, we do not use GitHub- or GitLab-specific terms to avoid confusion, as the architecture and workflow are identical for both. Furthermore, note that both components are independent of each other and can be replaced by other components that generate OpenAPI descriptions and compute the differences between them.

A. AutoGuard-gen

AutoGuard-gen reads the source code of a version of a web service implemented with Spring Boot and generates one OpenAPI description for each Spring profile.

1) AutoOAS: AutoGuard-gen uses AutoOAS [10], our approach for generating accurate OpenAPI descriptions directly from source code. Table I shows the Spring Boot annotations that AutoOAS supports. In the following, we provide a brief description of AutoOAS and refer the reader to [10] for more details.

AutoOAS consists of three stages. In the first stage, it parses and statically analyzes a Spring Boot project's source code using Spoon [12]. The outputs of this stage are the *controller* classes and *controller-advice* classes of the project. The controller classes contain all REST methods and the corresponding parameters and responses. The controller-advice classes contain logic about the service's global exception handling, which converts Java exceptions to HTTP response status codes. AutoOAS splits the controller classes based on Spring profiles and generates one OpenAPI description per profile.

[1]https://github.com/MSA-API-Management/AutoGuard

In the second stage, AutoOAS detects the REST methods, parameters, and responses in the subset of controller classes of the Spring profile and generates the OpenAPI *paths* section from them. Additionally, it uses the controller-advice classes to translate Java exceptions to HTTP responses status codes. It records all custom Java classes that are used as parameters or return values, i.e., the data models of the API. Finally, in the third stage, AutoOAS analyzes the Java classes of these data models and generates the OpenAPI *schemas* section.

We published AutoOAS as a Java command-line tool that requires two arguments: the Spring Boot project's source code directory *source_dir* and an output directory *output_dir* for storing the generated OpenAPI descriptions. Note, AutoOAS names the individual OpenAPI description files after the Spring profiles they describe.

2) AutoOAS evaluation: In our previous work [10], we evaluated AutoOAS on seven real-world Spring Boot projects and compared its precision and recall to three state-of-the-art approaches for generating OpenAPI descriptions, namely Respector [13], Prophet [14], and springdoc-openapi [15]. AutoOAS achieved the highest overall precision and recall for identifying REST methods (100% precision; 100% recall), parameters (73%; 69%), and responses (99%; 97%). Furthermore, it was the only approach correctly considering and representing the inheritance hierarchy in data models.

As part of our evaluation, we also analyzed the runtime performance of AutoOAS. It achieved an average runtime of 11.0 seconds and a median runtime of 7.6 seconds per project and took a maximum of 31.4 seconds for the largest project containing 28,099 lines of Java code. We argue that the runtime of AutoOAS is appropriate considering its high precision and recall as well as its accurate representation of data models compared to the other approaches. Due to its promising results and appropriate runtime, we selected AutoOAS for integration into AutoGuard.

```
1 docker run
2   -v <source_dir>:/project
3   alexx882/auto-oas:1.0
4   /project
5   /project/<output_dir>/oas
```

Listing 1. Shell command running the AutoOAS Docker image analyzing a Spring Boot project in *source_dir* and writing the generated OpenAPI descriptions to *output_dir* with the *oas* prefix.

3) Workflow integration: We provide a Docker image[2] of AutoOAS for integration into AutoGuard. The example command in Listing 1 runs the AutoOAS Docker image. It binds the *source_dir* to the */project* directory inside the Docker container (Line 2) and provides the same */project* directory as an argument to AutoOAS (Line 4). It writes the OpenAPI descriptions prefixed with *oas* to *output_dir* inside the */project* directory (Line 5).

AutoGuard-gen sets the *source_dir* to the repository's root directory and runs the Docker image. It persists the generated OpenAPI descriptions, temporarily stored in *output_dir*, on GitHub and GitLab as workflow and respectively job artifacts, making them available to developers and AutoGuard-diff.

B. AutoGuard-diff

AutoGuard-diff uses the OpenAPI descriptions generated for each Spring profile and version of the API to extract the differences between them. Then, it reports the differences and highlights API breaking changes.

1) OpenAPI-diff: AutoGuard-diff uses the OpenAPI-diff tool [11] to extract the differences between two OpenAPI descriptions. OpenAPI-diff compares the structure of the REST methods, parameters, and responses in the *paths* section and the data models in the *schemas* section between two OpenAPI versions and describes their differences in various output formats. Additionally, it classifies the identified changes as non-breaking and breaking based on backward compatibility. OpenAPI-diff is available as a Java library or a Docker image and takes the paths of two OpenAPI descriptions as input.

AutoGuard-diff tailors the execution of OpenAPI-diff with the following two arguments. First, it uses the `--html <file_path>` option to generate an HTML report of the identified differences. Figure 2 shows an example HTML change log of OpenAPI-diff. It describes a new REST method *POST /api/customers*, a deleted method *GET /api/customers/orders*, and changes to a parameter and the returned data model of the method *GET /api/customers/invoice*. Second, it uses the `--fail-on-incompatible` option to get feedback about breaking changes. With this option enabled, OpenAPI-diff will fail with an exit code of 1 if breaking changes are detected. If all changes are non-breaking, OpenAPI-diff returns the exit code 0.

2) Workflow integration: AutoGuard-diff takes as input two sets of OpenAPI descriptions for two REST API versions. Each set consists of the OpenAPI descriptions generated for each Spring profile of one API version. AutoGuard-diff pairs the OpenAPI descriptions between the sets based on their

[2]https://hub.docker.com/repository/docker/alexx882/auto-oas/general

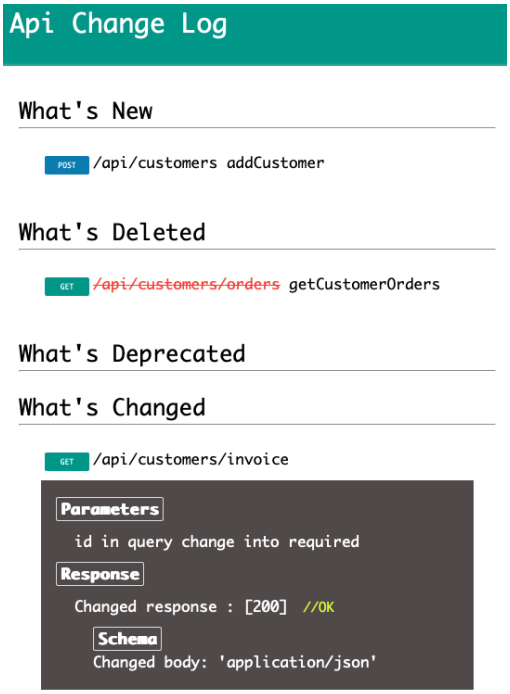

Fig. 2. HTML change log generated by OpenAPI-diff

file names, executes OpenAPI-diff for each file pair using the OpenAPI-diff Docker image with the options described above, and temporarily stores the change logs and exit codes.

After all OpenAPI description file pairs are analyzed, AutoGuard-diff persists the change logs as workflow and job artifacts on GitHub and GitLab, respectively. Finally, it sums up all exit codes, where a result of 1 or higher indicates that breaking changes were identified. In this case, AutoGuard-diff fails the whole AutoGuard workflow and reports the individual incompatible Spring profiles in the job log.

Notably, two edge cases lead to missing OpenAPI description files for one of the REST API versions. Deleting a Spring profile in the new version v_2 leads to a missing file for this profile in v_2, and adding a profile to a new version v_2 leads to a missing file for this profile in the previous version v_1. For both cases, AutoGuard-diff replaces the missing file with an empty OpenAPI description, i.e., without any *paths* or *schemas*. Consequently, AutoGuard only considers deleting a profile as a breaking change if the profile was not empty, i.e., if it contained REST methods in v_1 that are missing in the empty v_2. Adding a new profile introduces new REST methods, which are considered backward compatible [7] and therefore denoting a non-breaking change. Note, AutoGuard considers renamed profiles as added and deleted and consequently, reports a breaking change if the renamed profile contains REST methods. We argue that renaming a profile potentially requires changing run and deployment configurations and should not go unnoticed.

III. Breaking Change Scenario

In this section, we demonstrate how to integrate AutoGuard into an existing GitHub or GitLab repository. Our example

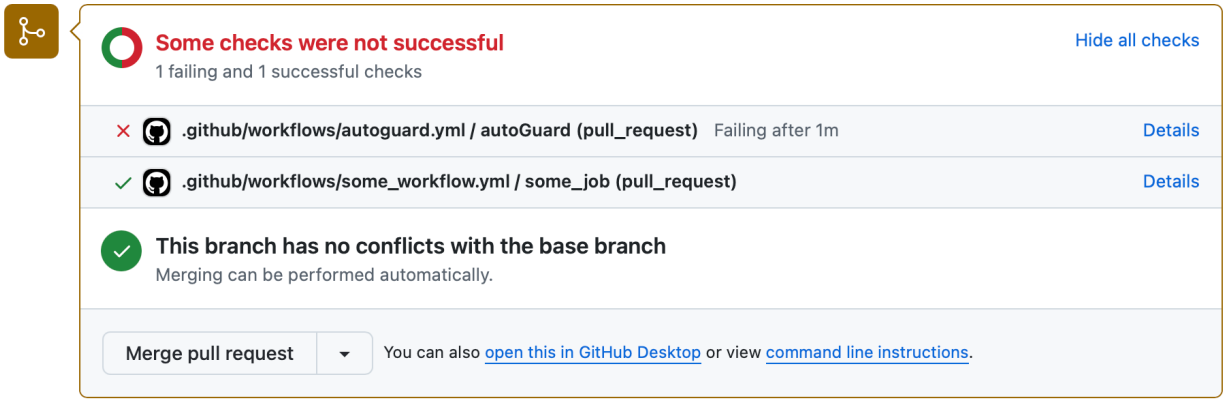

Fig. 3. The GitHub pull request with one failed check indicating that AutoGuard identified breaking API changes.

```
1 on: [pull_request]
2
3 jobs:
4   autoGuard:
5     runs-on: ubuntu-latest
6     steps:
7       - uses: MSA-API-Management/AutoGuard@v1
```

Listing 2. GitHub workflow running AutoGuard in the event of a pull request.

repository contains a simple Spring Boot web service with a single controller that consists of two REST methods for adding and querying *customer* data. Then, we show how AutoGuard extracts and reports a breaking API change from a source code change in the service's data model class. We provide the repository used for our scenario on GitHub[3].

A. GitHub and GitLab integration

The scenario starts with an existing repository containing the service's source code hosted on GitHub or GitLab.

1) GitHub: To use AutoGuard in a GitHub repository, the developer creates a new GitHub workflow and adds the AutoGuard GitHub action to it. Listing 2 shows the GitHub workflow in our example project. Line 1 configures the workflow to trigger on pull requests. Line 4 defines a job called *autoGuard* that runs the AutoGuard GitHub action (Line 7). With this configuration, AutoGuard automatically analyzes the head and base of any pull request, saves the OpenAPI descriptions and their differences as job artifacts, and fails the workflow if breaking changes are detected.

2) GitLab: To use AutoGuard in a GitLab repository, the developer creates or extends the GitLab pipeline and adds the AutoGuard GitLab template containing a job called *.autoGuard* to it. Listing 3 shows a configuration that triggers AutoGuard on merge requests in our example project. Lines 1 and 2 import the AutoGuard template to the configuration file, and Line 4 sets up the analysis for the repository's root directory. Lines 6 and 7 extend the *.autoGuard* job to define the execution stage (Line 10) and the rule to trigger the job on merge requests (Line 9). With this configuration, AutoGuard analyzes the source and target of any merge request, saves the OpenAPI descriptions and their differences as job artifacts, and fails the pipeline if breaking changes are detected.

[3]https://github.com/MSA-API-Management/AutoGuard-example-project

```
1 include:
2   - remote: 'https://raw.githubusercontent.com/␣
        MSA-API-Management/AutoGuard/refs/tags/v1/␣
        AutoGuard.gitlab-ci.yml'
3     inputs:
4       source_dir: "."
5
6 autoGuard:
7   extends: .autoGuard
8   rules:
9     - if: $CI_PIPELINE_SOURCE ==
          'merge_request_event'
10  stage: test
```

Listing 3. GitLab pipeline running AutoGuard in the event of a merge request.

B. Introducing a breaking change

After the repositories are set up, AutoGuard automatically guards pull and merge requests, respectively. In the following, we present this scenario for GitHub. Once a developer opens a pull request to the main branch in our example project, the AutoGuard workflow is triggered. First, AutoGuard-gen generates the OpenAPI descriptions for the pull request head on the feature branch and the pull request base on the main branch. Next, AutoGuard-diff identifies the OpenAPI description pairs for each profile and extracts the differences between them.

In our example scenario, the feature branch introduces an unexpected breaking change to the REST API by adding a `@NotNull` annotation to the customer's email field. Accordingly, AutoGuard detects this change and fails the workflow with the information that the pull request would break the API's backward compatibility. We show the corresponding GitHub pull request in Figure 3. With this information, the developers and code reviewers can make informed decisions on how to proceed with the pull request.

IV. Related Work

Multiple related works focused on detecting and classifying source code and API changes. On the source code level, existing approaches [16], [17] use control and data flow analysis to identify semantic changes and their impacts inside a component. However, they do not consider the impact of

code changes on the API level. Several approaches [18], [19] detected changes in APIs of web services specified in the Web Service Description Language (WSDL). However, they can not handle REST APIs, which have become the standard for implementing communication between web services [7].

OpenAPI-diff [11] identifies differences between REST APIs described in the OpenAPI Specification. Similarly, Schmiedmayer et al. [20] identify and classify changes between OpenAPI descriptions. As a limitation, they integrated their approach into a Swift framework. Serbout et al. [21] used OpenAPI descriptions to visualize the evolution of REST APIs over time, thus aggregating individual changes or losing temporal context. All these approaches require the availability of OpenAPI descriptions during analysis.

Previous approaches and Swagger tools [15], [22]–[24] used runtime reflection to generate the necessary OpenAPI descriptions and, hence, required the domain knowledge to create a valid run configuration for the service and the infrastructure to run it.

To the best of our knowledge, we propose the first tool for reporting individual breaking changes in REST APIs that is directly integratable into the development process. AutoGuard uses the static OpenAPI generation capabilities of AutoOAS [10], and hence, does not require executing any service under analysis to identify its REST API changes.

V. Conclusion

In this work, we presented AutoGuard, our tool for extracting and reporting breaking changes of REST APIs directly from Java Spring Boot source code changes. AutoGuard uses AutoOAS to generate the OpenAPI descriptions of two versions of a REST API and OpenAPI-diff to identify their non-breaking and breaking changes. AutoGuard does not require running any service under analysis, making it applicable for direct integration into the development process. We integrated AutoGuard into the CI workflows of GitHub and GitLab, where it automatically generates the REST API change logs for pull (or merge) requests and indicates breaking changes by failing the workflow (or pipeline). We provide the GitHub action, GitLab template, and an example project on GitHub.

As limitations, AutoGuard only supports web services developed with the Java Spring Boot framework and reports changes based on OpenAPI descriptions, which do not capture semantic changes, e.g., changed parameter dependencies [25].

In future work, we plan to evaluate AutoGuard in a user study. Furthermore, we plan to extend AutoGuard-gen to support other frameworks, such as Jersey [26], and to improve AutoGuard-diff, for instance, to report individual data model changes in the change log.

References

[1] R. T. Fielding, "Rest: architectural styles and the design of network-based software architectures," *Doctoral dissertation, University of California*, 2000.

[2] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017.

[3] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Incorporated, 2019.

[4] K. Gos and W. Zabierowski, "The comparison of microservice and monolithic architecture," in *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pp. 150–153, 2020.

[5] S.-P. Ma, C.-Y. Fan, Y. Chuang, I.-H. Liu, and C.-W. Lan, "Graph-based and scenario-driven microservice analysis, retrieval, and testing," *Future Generation Computer Systems*, vol. 100, pp. 724–735, 2019.

[6] O. Zimmermann, M. Stocker, D. Lübke, C. Pautasso, and U. Zdun, "Introduction to microservice api patterns (map)," in *International Conference on Microservices (Microservices 2019)*, 2019.

[7] A. Lercher, J. Glock, C. Macho, and M. Pinzger, "Microservice api evolution in practice: A study on strategies and challenges," *Journal of Systems and Software*, vol. 215, p. 112110, 2024.

[8] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: Current and future directions," *SIGAPP Appl. Comput. Rev.*, vol. 17, p. 29–45, jan 2018.

[9] Spring, "Spring Boot." https://spring.io/projects/spring-boot. [Accessed 10-10-2024].

[10] A. Lercher, C. Macho, C. Bauer, and M. Pinzger, "Generating accurate openapi descriptions from java source code," *arXiv*, 2024. https://doi.org/10.48550/arXiv.2410.23873.

[11] OpenAPITools, "OpenAPI-diff." https://github.com/OpenAPITools/openapi-diff. [Accessed 10-10-2024].

[12] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015.

[13] R. Huang, M. Motwani, I. Martinez, and A. Orso, "Generating rest api specifications through static analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[14] T. Cerny, A. S. Abdelfattah, J. Yero, and D. Taibi, "From static code analysis to visual models of microservice architecture," *Cluster Computing*, vol. 27, pp. 4145–4170, Jul 2024.

[15] springdoc, "springdoc-openapi." https://github.com/springdoc/springdoc-openapi. [Accessed 10-10-2024].

[16] Q. Hanam, A. Mesbah, and R. Holmes, "Aiding code change understanding with semantic change impact analysis," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 202–212, 2019.

[17] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 819–830, 2019.

[18] A. Chaturvedi and D. Binkley, "Web service slicing: Intra and inter-operational analysis to test changes," *IEEE Transactions on Services Computing*, vol. 14, no. 3, pp. 930–943, 2021.

[19] D. Romano and M. Pinzger, "Analyzing the evolution of web services using fine-grained changes," in *2012 IEEE 19th International Conference on Web Services*, pp. 392–399, 2012.

[20] P. Schmiedmayer, A. Bauer, and B. Bruegge, "Reducing the impact of breaking changes to web service clients during web api evolution," in *2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 1–11, 2023.

[21] S. Serbout, D. C. Muñoz Hurtado, and C. Pautasso, "Interactively exploring api changes and versioning consistency," in *2023 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 28–39, 2023.

[22] S. Serbout, A. Romanelli, and C. Pautasso, "Expresso: From express.js implementation code to openapi interface descriptions," in *Software Architecture. ECSA 2022 Tracks and Workshops* (T. Batista, T. Bureš, C. Raibulet, and H. Muccini, eds.), (Cham), pp. 29–44, Springer International Publishing, 2023.

[23] springfox, "SpringFox." https://github.com/springfox/springfox. [Accessed 10-10-2024].

[24] Swagger, "Swagger Core." https://github.com/swagger-api/swagger-core. [Accessed 10-10-2024].

[25] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "A catalogue of inter-parameter dependencies in restful web apis," in *Service-Oriented Computing* (S. Yangui, I. Bouassida Rodriguez, K. Drira, and Z. Tari, eds.), (Cham), pp. 399–414, Springer International Publishing, 2019.

[26] Eclipse Foundation, "Eclipse Jersey." https://eclipse-ee4j.github.io/jersey/. [Accessed 10-10-2024].