The Journal of Systems and Software 244 (2027) 113122

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Generating accurate OpenAPI descriptions from Java source code☆

Alexander Lercher *, David Jamnig, Christian Macho, Clemens Bauer, Martin Pinzger

Department of Informatics Systems, University of Klagenfurt, Universitätsstraße 65-67, Klagenfurt, 9020, Austria

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.18877085

Keywords:
OpenAPI specification
Source code analysis
REST API

ABSTRACT

Developers require accurate descriptions of REpresentational State Transfer (REST) Application Programming Interfaces (APIs) for implementing successful interactions between web services. The OpenAPI Specification (OAS) has become the de facto standard for documenting REST APIs. Manually creating an OpenAPI description is time-consuming and error-prone, and therefore, several approaches were proposed to automatically generate them from source code or runtime information.

In this paper, we first study four such state-of-the-art approaches, Respector, Prophet, springdoc-openapi, and Swagger Core, and discuss their shortcomings. Next, we introduce AutoOAS, our approach addressing these shortcomings to generate accurate OpenAPI descriptions. It detects exposed REST API endpoint paths, corresponding HTTP methods, HTTP response codes, and the data models of request parameters and responses directly from Java source code.

We evaluated AutoOAS on 15 real-world Java projects developed with the two popular frameworks Spring Boot and JAX-RS and compared its performance with the four state-of-the-art approaches. Based on a manually created ground truth, AutoOAS achieved the highest precision and recall when identifying REST API endpoint paths, HTTP methods, parameters, and responses. It outperformed the second-best approach, Respector, by 29 percentage points in precision and 27 in recall when identifying parameters, and by 25 percentage points in precision and 34 in recall when identifying responses. Furthermore, AutoOAS generated more accurate OpenAPI data model schemas than the framework-specific approaches springdoc-openapi and Swagger Core, outperforming them by 30 and 14 percentage points, respectively.

1. Introduction

REpresentational State Transfer (REST) (Fielding, 2000) Application Programming Interfaces (APIs) are widely used for communication between web services. Due to their loose coupling, service developers require documentation of the REST APIs to correctly implement API calls to other services. The OpenAPI Specification (OAS) (OpenAPI Initiative, 2026d) serves as the de-facto standard for describing REST APIs, and the resulting OpenAPI descriptions are typically shared with other development teams (Lercher et al., 2024).

An *OpenAPI description* documents a REST API in two sections. The REST API endpoint paths and corresponding HTTP methods are located in the OAS section *#/paths*, and the data models for the REST API's parameters and responses are contained in the section *#/components/schemas*. The # represents the OpenAPI description's root. In the remainder of this paper, we use the term *operation* when referring to the unique combination of REST API endpoint path and HTTP method, and the term *handler* when referring to the source code method handling the REST API call to clearly distinguish them.

Various tools (APIs You Won't Hate, 2026) and research approaches (Koren and Klamma, 2018; Peng et al., 2018; Ed-Douibi et al., 2020; Karlsson et al., 2020; Arcuri, 2019; Kim et al., 2024) use OpenAPI descriptions, e.g., for visualization, testing, and code generation, and, consequently, rely on accurate descriptions. However, conventional approaches for obtaining an OpenAPI description have various challenges. Manually creating OpenAPI descriptions is time-consuming, error-prone, and increases the risk that a service's source code and OpenAPI description diverge (Lercher et al., 2024). Many automated approaches (springdoc, 2026a; Swagger, 2026a; Yandrapally et al., 2023; Serbout et al., 2023) require running the web service under analysis to generate its OpenAPI description. Hence, they require the domain knowledge to create a valid run configuration for the service and the infrastructure resources to run it.

Recently, two static analysis approaches emerged that generate REST API descriptions from Java code. Huang et al. (2024) proposed Respector for generating OpenAPI descriptions from bytecode of web

☆ Editor: Dario Di Nucci.

* Corresponding author.
E-mail address: alexander.lercher@aau.at (A. Lercher).

https://doi.org/10.1016/j.jss.2026.113122
Received 19 March 2026; Received in revised form 16 September 2026; Accepted 24 September 2026
Available online 30 September 2026
0164-1212/© 2026 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

services created with Spring Boot (Spring, 2026e) and Jakarta RESTful Web Services (JAX-RS) (Eclipse Foundation, 2026b). Respector also symbolically analyzes the bytecode to infer REST API parameter constraints, which it encodes in the OpenAPI description using custom extensions. They compared Respector to AppMap (AppMap, 2026), Swagger Core (Swagger, 2026a), SpringFox (springfox, 2026), and springdoc-openapi (springdoc, 2026a) and outperformed all four state-of-the-art tools for generating API operations, parameters, and responses. Cerny et al. (2024) proposed Prophet for extracting REST APIs from the source code of Java Spring Boot projects. Prophet does not generate an OpenAPI description but creates a custom JSON output used as an intermediate format for visualizing microservice dependency graphs. This format can be easily converted into an OpenAPI description.

However, we identified several limitations of the existing static analysis approaches, Respector and Prophet. First, they do not consider Spring's runtime profiles (Spring, 2026d) and miss several framework annotations used to specify the REST API. Second, they fail to correctly translate thrown exceptions to dedicated HTTP response status codes. Third, they do not accurately document the data models of a REST API, which is a requirement for the various tools that build on OpenAPI descriptions.

In this paper, we aim to understand and address the shortcomings of existing approaches and present AutoOAS, our static analysis approach for generating accurate and detailed OpenAPI descriptions. For this, we address the following research questions (RQs):

RQ1 What are the shortcomings of state-of-the-art approaches for generating OpenAPI descriptions?
RQ2 How accurately does AutoOAS detect REST API operations, parameters, and response codes from Java source code compared to state-of-the-art approaches?
RQ3 How accurately does AutoOAS represent REST APIs as OpenAPI descriptions compared to state-of-the-art approaches?
RQ4 What is the runtime performance of AutoOAS compared to state-of-the-art approaches?

AutoOAS extracts REST API operations, parameters, responses, and data models directly from the source code of web services created with the popular Java frameworks Spring Boot and JAX-RS. It accounts for runtime profiles and exception handling, and accurately represents data models, including inheritance information. We compare AutoOAS to Respector (Huang et al., 2024), Prophet (Cerny et al., 2024), springdoc-openapi (springdoc, 2026a), and Swagger Core (Swagger, 2026a), which are the state-of-the-art approaches for extracting the OpenAPI description from bytecode, source code, and runtime reflection. For our evaluation, we extended an existing dataset (Huang et al., 2024) containing seven Spring Boot and eight JAX-RS projects. We corrected and extended the dataset's ground truth of API operations, parameters, and response codes by manually extracting the information from the source code, and manually created OpenAPI descriptions for a subset of each project's operations. The evaluation results show that AutoOAS obtained the highest precision and recall for identifying operations, parameters, and response codes, and it produced the most accurate schema descriptions. In summary, this paper makes the following contributions:

- An improved evaluation dataset describing the REST APIs of seven Java Spring Boot and eight JAX-RS projects.
- An analysis of the shortcomings of the state-of-the-art approaches for generating OpenAPI descriptions.
- Our approach, AutoOAS, addressing the shortcomings and thereby generating more accurate OpenAPI descriptions.

The remainder of this paper is structured as follows. Section 2 describes the evaluation dataset and the improvements that we performed on it. Section 3 discusses the shortcomings of existing approaches for generating OpenAPI descriptions. Our approach, AutoOAS, which addresses these shortcomings, is presented in Section 4. In Section 5, we evaluate precision and recall of AutoOAS for detecting API operations, parameters, and response codes compared to existing approaches. We evaluate the representation accuracy of the generated OpenAPI descriptions in Section 6. In Section 7, we evaluate the runtime performance of AutoOAS. Section 8 discusses the results, implications, and threats to validity, and Section 9 presents related work. Finally, Section 10 concludes the paper.

2. Improved evaluation dataset

For our evaluations, we built upon the dataset curated by Huang et al. (2024) containing seven Java Spring Boot projects, eight JAX-RS projects, and a document reporting their ground truth (GT document) containing the projects' API operations, parameters, and response codes. The dataset contains source code snapshots of the following projects: CatWatch (The Zalando Incubator, 2026), Corona-Warn-App Verification Server (CWA) (Deutsche Telekom AG, 2026), Open Contracting Vietnam (OCVN) (Development Gateway, 2026), Ohsome API (GIScience Research Group and HeiGIT, 2026), ProxyPrint (ProxyPrint, 2026), Quartz Manager (Formosa, 2026), Ur-Codebin Mathew Estafanous (2026), Cassandra (DataStax Inc., 2026), Digdag (Treasure Data, 2026), EnviroCar (The enviroCar Project, 2026), Features-service (Ferrara, 2026), Gravitee (Gravitee.io, 2026), Kafka (Confluent Inc., 2026), RESTcountries (APILayer, 2026), and Senzing (Senzing community, 2026).

The GT document provides a compact representation of the projects' API operations, parameters, and response codes that is independent of OpenAPI versions and constructs. This document serves as the foundation for answering RQ1 and RQ2. However, we identified multiple inconsistencies and shortcomings in the GT document during our evaluations of the existing approaches and AutoOAS. Accordingly, we manually validated the GT document against the source code and improved the original version to create our improved ground truth (GT+) document.

Moreover, we manually created OpenAPI descriptions for a subset of each project's API operations to capture the semantic structure of the corresponding parameters, responses, and data models. These OpenAPI descriptions complement the GT+ document by serving as the ground truth for representing OpenAPI structures and constructs. We use the GT+ OpenAPI descriptions to answer RQ3.

In the following, we describe the identified inconsistencies and the improvements we performed to create our GT+ document and the GT+ OpenAPI descriptions. Table 1 lists the projects and descriptive statistics of the improved dataset, which is available as part of our replication package (Lercher et al., 2026).

2.1. Improvements to the GT document

Huang et al. (2024) published the original GT document as an Excel file containing one sheet per project that lists the corresponding API operations, parameters, and responses. We retained this structure for the GT+ document.

The improvements to the GT document followed a predefined protocol. First, we executed Respector, Prophet, and AutoOAS on each project and compared the analysis results to the GT to identify potential inconsistencies in the GT document. We tracked each reported false positive and false negative case for API operations, parameters, and responses. We clarify that the analysis results of the three approaches served solely as indicators of potential inconsistencies between the GT and the source code and were not used to revise the GT document.

For each project, one author manually validated each potential inconsistency in the GT document against the project's source code. If the GT did not correctly represent the REST API defined in the source code, i.e., the reported false positive or false negative case was

Table 1
Descriptive statistics of the seven Spring Boot and eight JAX-RS projects.

Project name	Framework	Java LoC	# API operations	# API parameters	# API responses
CatWatch	Spring Boot	6454	14	36	19
CWA	Spring Boot	3616	6	16	16
OCVN	Spring Boot	28,099	278	5,018	278
Ohsome	Spring Boot	10,597	159	1937	179
ProxyPrint	Spring Boot	6052	75	157	103
Quartz	Spring Boot	3883	14	15	20
Ur-Codebin	Spring Boot	1962	6	14	15
Cassandra	JAX-RS	8204	50	95	123
Digdag	JAX-RS	90,435	41	97	87
EnviroCar	JAX-RS	25,495	128	449	507
Features-service	JAX-RS	1488	18	35	29
Gravitee v4	JAX-RS	188,671	28	110	47
Kafka	JAX-RS	45,656	69	193	124
RESTcountries	JAX-RS	1605	27	37	72
Senzing	JAX-RS	50,892	34	164	103

Table 2
Snippet of the GT+ document for two operations in project EnviroCar.

path	method
/groups/{group}	GET
/groups/{group}	PUT

Table 3
Snippet of the GT+ document with the parameters for *PUT /groups/{group}* in project EnviroCar.

path	method	type	name
/groups/{group}	PUT	string	group
/groups/{group}	PUT	string	description
/groups/{group}	PUT	string	name
/groups/{group}	PUT	User	owner
/groups/{group}	PUT	string	creationTime
/groups/{group}	PUT	string	modificationTime

in fact a true positive or true negative, the author updated the GT with the actual operation, parameter, or response from the source code. In addition, the same author sampled a quarter of the GT document entries that were not contained in the set of potential inconsistencies and compared them with the project's source code. Vice versa, the author manually inspected a subset of API operations, parameters, and responses in the project's source code and cross-checked them with the GT document. If either complementary check revealed further inconsistencies, the author updated the GT document with the actual operations, parameters, or responses defined in the source code and investigated the project more closely by repeating the cross-checking process with an additional sample.

We versioned the GT throughout the study to ensure traceability of the performed changes. After one author altered the GT document, the proposed updates were subsequently cross-checked by a different author and either accepted or rejected. The two authors discussed all rejected changes until they reached a final decision to either accept or reject each proposed change.

In the following, we describe the improvements incorporated into the GT+ document.

2.1.1. API operations

The API operations section of the GT+ document consists of two columns representing the endpoint paths and methods. Table 2 shows an excerpt containing two operations from project EnviroCar.

We identified several inconsistencies in the original GT document. The CWA project consists of two Spring profiles called *external* and *internal*. Spring profiles (Spring, 2026d) allow developers to segregate parts of the web service and its REST API at runtime. In particular, Spring provides an `@Profile` class annotation assigning the corresponding class to one or multiple profiles. At runtime, it only instantiates the classes of active profiles based on the service configuration. In project CWA, both profiles contain the same operation and, due to Spring Boot's constraint of unique operations, the two profiles cannot be simultaneously active at runtime. The GT missed this constraint. We improved it by creating two separate service descriptions to consider this runtime behavior in our GT+.

Similarly, the JAX-RS project Gravitee contains two versions of the API, *v1* and *v4*, which expose the same operations and share the same data models. Again, they cannot be simultaneously active at runtime. The GT contained partial information of both versions but neither was complete. Due to the project's size, we decided to only obtain the GT+ for version *v4*, which is also the one analyzed by state-of-the-art approaches.

Moreover, the GT contained incorrect operation paths for three projects. In particular, we found that all paths in project Ur-Codebin were missing an */api* prefix, all paths in project RESTcountries missed a */rest* prefix, and many paths in project Kafka were missing a */v3* prefix. We fixed these paths in the GT+ document.

2.1.2. API parameters

We noticed that the GT does not contain parameter locations, e.g., path or query. Furthermore, it flattens object parameters into multiple individual parameters based on the object's fields. For instance, a single object parameter with five fields results in five individual parameters. Table 3 shows the GT document's representation of a path parameter `group` and a request body object consisting of five fields. We maintain this structure of the GT document for evaluating the correct detection of parameters and separately evaluate the parameter locations and data model representation of objects with the GT+ OpenAPI descriptions.

The GT missed several request body object fields in project Gravitee v4 and many more in project EnviroCar. Notably, the OAS does not describe request body names because they are transmitted as anonymous JSON objects in REST. Correspondingly, we renamed and added the field names as parameters in the GT+ and thereby adopted the flat representation of the GT.

Next, the GT missed multiple request body parameters for the projects CatWatch, Quartz, Digdag, Kafka, RESTcountries, and Senzing. After verifying their appearances in the source code, we added all of them to the GT+.

In project CatWatch, we discovered that the GT incorrectly reported a query parameter name as `sort_by` instead of `sortBy`. Similarly, the GT contained incorrect path parameters for the projects EnviroCar, Gravitee v4, Kafka, and RESTcountries. For instance, in project EnviroCar, the GT reported path parameter names as `trackId` instead of `track`. We updated the parameter names in our GT+ accordingly.

Table 4
Snippet of the GT+ document with the responses for *GET /groups/{group}* in project EnviroCar.

path	method	status code	type
/groups/{group}	GET	200	object
/groups/{group}	GET	401	empty
/groups/{group}	GET	403	empty
/groups/{group}	GET	404	empty

2.1.3. API response codes

The API response section of the GT document describes a response as a tuple of the status code and an abstracted return type, e.g., string, array, object, or empty. For example, Table 4 shows an excerpt with the responses for one operation in project EnviroCar. Similarly to API parameters, we decided to retain the GT document structure for evaluating the correct detection of status codes and separately evaluate the correctness of their corresponding response schemas against the GT+ OpenAPI descriptions.

We introduced multiple improvements to the response section of the GT+ document for two reasons: the original GT reported wrong default response codes and did not contain any error responses.

The GT assumes that handlers with the void return type and handlers returning null translate to the HTTP response code *204 No Content*. However, Spring Boot returns *200 OK* by default and only returns *204* if the status code is explicitly set. We found and fixed this inconsistency in the projects OCVN, Ohsome, and ProxyPrint. In JAX-RS, the default HTTP response code for void handlers is *204 No Content*. As exceptions, project Kafka uses asynchronous responses and project Senzing uses server-sent events, both of which return *200 OK* for void handlers by default. While the GT reflects the behavior of regular void handlers and server-sent events in JAX-RS correctly, it incorrectly assumes *204 No Content* for asynchronous void handlers in project Kafka. We updated our GT+ for this project accordingly. Moreover, the GT incorrectly reported *206 Partial Content* for multiple operations in project EnviroCar that allow pagination. After inspecting the source code, we found that they return the default response code *200 OK*, which we corrected in our GT+.

The GT only reports successful HTTP response codes, i.e., in the *2xx* range. It does not contain other response codes that the projects return directly or by throwing exceptions. However, both Java frameworks support translating uncaught Java exceptions in handlers to HTTP response codes. Spring Boot provides the `@ExceptionHandler` method annotation for this translation. The annotated method must either be located in the same class as the handler or in another class which is annotated as `@ControllerAdvice`. JAX-RS offers the `@Provider` class annotation in combination with the `ExceptionMapper` interface to translate uncaught exceptions. Furthermore, JAX-RS provides multiple exceptions that directly translate to response codes, e.g., `BadRequestException`.

To capture redirection responses, i.e., *3xx* range, and error responses, i.e., *4xx* and *5xx* ranges, we inspected the source code of each project and, starting from the handler, documented each encountered response code until we reached a method call to an interface reference. We considered response codes that were either set directly, e.g., with Spring's `ResponseEntity::status` method, translated from exceptions automatically, e.g., JAX-RS's `BadRequestException`, translated from exceptions manually, e.g., by implementing an `ExceptionMapper`, or *500 Internal Server Error* resulting from untranslated exceptions. Thereby, we found additional response codes for all projects except OCVN and extended the GT+ accordingly.

2.2. GT+ OpenAPI descriptions

To evaluate the representation accuracy of API parameters, responses, and data models, we manually created OpenAPI descriptions for a subset of each project's API operations based on the corresponding source code. The creation process followed a predefined protocol.

First, we randomly sampled ten operations per project from the GT+ document. Since the projects CWA and Ur-Codebin each contain only six operations, we included all their operations for analysis. We sampled the operations because manually creating and validating detailed OpenAPI descriptions requires substantial effort, while describing all operations would result in many repeated instances of the same OpenAPI constructs. During the manual documentation process, we inspected the source code of each project, including all call chains from the handlers to the point where we reached a call to an interface reference.

One author analyzed the source code of all sampled operations in each project and generated the corresponding OpenAPI descriptions. Furthermore, we randomly selected two of the 15 projects for independent analysis by two additional authors, and randomly selected two more projects for analysis by one of the additional authors each. In sum, the three authors independently created the OpenAPI descriptions for the projects Gravitee v4 and Ohsome three times and for the projects ProxyPrint and RESTcountries twice based on the source code. They then compared the OpenAPI descriptions and discussed all differences until the involved authors agreed on how to resolve each difference. The script for comparing OpenAPI descriptions is available in our replication package (Lercher et al., 2026).

We fully described the operations, parameters, request body media types, response codes, response headers, and response body media types in the OAS *#/paths* section. Furthermore, we modeled complex data model schemas and schema inheritance as explicitly named schemas in the OAS *#/components/schemas* section and referenced these schemas from the corresponding request and response bodies. In total, we manually described 142 API operations, 432 API parameters (min=2; med=15.5; max=125), 331 API responses (min=7; med=18.5; max=46), and 308 named data model schemas (min=1; med=14; max=71). Table 5 provides descriptive statistics of the 16 GT+ OpenAPI descriptions.

Relative to the full GT+ document, the 16 GT+ OpenAPI descriptions in total cover 142 out of 947 operations (15.0%), 571 out of 8373 simple parameters and object parameter fields (6.8%), and 331 out of 1722 responses (19.2%). For individual GT+ OpenAPI descriptions, coverage ranges from 3.6% to 100.0% of operations (med=32.6%), from 2.1% to 100.0% of simple parameters and object parameter fields (med=30.9%), and from 3.6% to 100.0% of responses (med=34.8%).

3. Shortcomings of existing approaches

In this section, we use the GT+ to evaluate four state-of-the-art approaches for generating OpenAPI descriptions, namely Respector (Huang et al., 2024), Prophet (Cerny et al., 2024), springdoc-openapi (springdoc, 2026a), and Swagger Core (Swagger, 2026a) to answer RQ1. While Respector supports both Java frameworks, Spring Boot and JAX-RS, Prophet and springdoc-openapi are limited to Spring Boot, and Swagger Core only supports JAX-RS.

3.1. Evaluation setup

We generated the OpenAPI descriptions for each project with each approach and compared them to the GT+ document. Furthermore, we manually evaluated the data model representation accuracy based on whether approaches used the OpenAPI *#/components/schemas* section and correctly described OpenAPI schemas compared to the respective Java classes.

3.1.1. Respector setup

For Respector, we compiled the source code of each project to obtain the bytecode necessary for its analysis. We used the compilation script included in Respector's replication package to ensure that we evaluated the same projects' bytecode as the original Respector evaluation.

Table 5
Number of API operations, parameters, responses, and data model schemas documented with the GT+ OpenAPI descriptions.

Project name	Ops	Params	ReqBodyMT	Resp	RespHead	RespBodyMT	Schemas	Obj	Derived	Props	Refs
CatWatch	10	20	1	14	0	8	9	9	0	74	2
CWA (external)	3	3	3	7	0	3	5	4	0	11	1
CWA (internal)	3	2	2	9	1	2	6	5	1	10	0
OCVN	10	106	0	10	0	9	39	30	2	165	39
Ohsome	10	125	0	30	9	29	16	15	3	47	11
ProxyPrint	10	10	3	18	0	9	17	16	2	64	9
Quartz	10	2	2	14	0	6	12	10	3	32	5
Ur-Codebin	6	6	2	15	0	5	7	4	0	17	5
Cassandra	10	4	1	23	0	26	1	1	0	7	0
Digdag	10	17	3	26	0	11	18	13	0	57	32
EnviroCar	10	26	3	46	3	7	19	11	6	70	16
Features-service	10	17	1	16	1	5	4	4	0	14	1
Gravitee v4	10	14	1	23	8	18	49	33	2	189	44
Kafka	10	25	2	19	0	8	26	19	7	71	20
RESTcountries	10	12	0	25	0	25	9	9	3	51	2
Senzing	10	43	11	36	0	38	71	60	19	191	39
Total	142	432	35	331	22	209	308	243	48	1070	226

Ops=Operations, Params=Parameters, ReqBodyMT=Request body media types,
Resp=Responses, RespHead=Response headers, RespBodyMT=Response body media types,
Schemas=Named schemas, Obj=Named object schemas, Derived=Named derived schemas,
Props=Named object schema properties, Refs=References in named object schema properties.

3.1.2. Prophet setup

Prophet generates a custom JSON format as analysis output. As an example, Listing 1 shows an operation description from Prophet for project CatWatch. The analysis results contain the endpoint path (Line 4), HTTP method (Line 1), parameters (Line 2), and success response (Line 3). The output also contains information not relevant to OpenAPI descriptions, such as source code class names and file paths. However, Prophet does not report the actual parameter and response schemas but only the string literals in the Java method signatures.

```
1 { "httpMethod": "GET",
2   "arguments": "[@RequestBody(required␣=␣false)␣String␣
        scoringProject,␣@RequestHeader(value␣=␣\"X-
        Organizations\",␣required␣=␣false)␣String␣
        organizations]",
3   "returnType": "java.lang.String",
4   "path": "/config/scoring.project",
5 [...] }
```

Listing 1: A snippet of Prophet's output describing one operation of CatWatch.

For our evaluation, we created a conversion script to convert Prophet's custom JSON format into OpenAPI descriptions. First, the conversion script directly copies the endpoint path and HTTP method from the original output. Then, it analyzes each parameter in the output list to identify its Java annotations, type, and name, and creates the according OpenAPI parameters section. The script analyzes the parameter annotations to report request body parameters in the corresponding OpenAPI section. We highlight that Prophet does not identify any response codes, but only the Java method's return type. Correspondingly, the conversion script applies the special OAS response code *default* (OpenAPI Initiative, 2026e), indicating no specific information, and reports Prophet's `returnType` as response content. Note that the converted output contains no additional schema information but only the Java method parameter names and types extracted from Prophet's string literals. We provide the conversion script for Prophet in our replication package (Lercher et al., 2026).

3.1.3. Springdoc-openapi and Swagger Core setup

The two runtime approaches use Java reflection to analyze framework annotations and expose the resulting OpenAPI descriptions as additional API operations. Hence, they require running the projects under analysis.

For each project, we first attempted to identify a valid run configuration and then to add a compatible springdoc-openapi or Swagger Core dependency to the project configuration. If a project depended on database connections or additional services, we attempted to start the required components whenever they were available as Docker images. Following this procedure, we were able to generate the OpenAPI descriptions of five Spring Boot and five JAX-RS projects.

We successfully analyzed the Spring Boot projects CatWatch, CWA, Ohsome, and Ur-Codebin with springdoc-openapi version 1.8, and project Quartz with springdoc-openapi version 1.5. We were unable to execute project OCVN because it depends on Node.js v6, which reached end of life and could not be linked successfully. For project ProxyPrint, we could not identify a springdoc-openapi version that is compatible with the existing project dependencies and versions.

For JAX-RS, we successfully analyzed the projects Cassandra, Features-service, Gravitee v4, Kafka, and RESTcountries with Swagger Core version 2.2. We could not identify a Swagger Core version for project Digdag that is compatible with the project's existing dependencies. Furthermore, we were unable to execute project EnviroCar because it required an extensive number of external components at startup, and project Senzing because it depends on a dynamically linked library that is not openly available.

3.2. API operations

Existing approaches did not always extract operations correctly because of the following reasons.

3.2.1. Spring profiles

Project CWA uses Spring profiles to distinguish its runtime behavior (cf. Section 2.1.1). The two profiles named *external* and *internal* each contain a handler exposing the same operation but different parameters, response codes, and even a different response body.

Respector does not support Spring Boot profiles and only generated a single OpenAPI description containing one of the two duplicate operations and all other operations of both profiles. Prophet correctly detected the duplicate operation but reported all operations in the same file without information about their accessibility at runtime. Notably, the OAS prohibits duplicate operations in an OpenAPI description, and Prophet could only report both in the same file because of its custom output format. springdoc-openapi inherently generated two separate OpenAPI descriptions because we started the service twice, once for each profile configuration. It detected the correct subset of operations for the active profile because only these handlers are loaded at startup.

3.2.2. JAX-RS's ApplicationPath annotation

Project RESTcountries uses the `@ApplicationPath` annotation, which, when annotating the main `Application`, defines a global prefix for all endpoint paths. Respector ignored this annotation and the corresponding */rest* prefix for all operations. Swagger Core, the second approach that supports JAX-RS projects, correctly reported the paths with the global prefix.

3.2.3. Constants in paths

Prophet could not resolve constants in endpoint paths but instead reported the string literal as the path, e.g., */api/ENDPOINT_NAME*. This affected the projects CWA and Quartz, which extensively use constants.

3.2.4. Regular expression constraints

Project OCVN uses regular expressions for several path parameters, encoded in the path string as displayed in Listing 2 Line 2. Respector correctly generated the path but ignored the constraint in the parameter description. Similar to handling constants, Prophet wrongly retained the literal path string as the path.

```
1 @RequestMapping(
2     value = "/api/ocds/organization/procuringEntity/id
          /{id:^[a-zA-Z0-9]*$}",
3     method = { RequestMethod.POST, RequestMethod.GET },
4     produces = "application/json")
```

Listing 2: Spring Boot `@RequestMapping` annotation containing a path parameter *id* with a regular expression constraint.

3.2.5. Spring's RequestMapping annotation

Prophet incorrectly reported the API operations for handlers annotated with Spring Boot's `@RequestMapping` in multiple projects. Sometimes, it identified only one HTTP method when the annotation defined multiple methods, e.g., it missed the *GET* HTTP method in Listing 2 Line 3. Notably, Prophet did not consistently report only the first method in the list. It also misclassified some HTTP methods as *GET* when the annotation defined another one. Furthermore, Prophet missed all `@RequestMapping` annotations which did not set any HTTP method, e.g., in project ProxyPrint. In this case, Spring Boot exposes the endpoint path with all HTTP methods, which was correctly reported by Respector.

3.3. API parameters

We found the following reasons for misidentified parameters.

3.3.1. Spring's ModelAttribute annotation

Project OCVN extensively uses the `@ModelAttribute` annotation, which maps individual parameters to a single Java object. After inspecting Respector's OpenAPI description, we discovered that it reported such parameters as a single request body object. However, Spring Boot does not bind the request body of the HTTP request to a model attribute object when receiving an API call (Spring, 2026c). Instead, it expects one path, query, or form parameter per field and builds the Java object correspondingly. Hence, API calls according to Respector's OpenAPI description fail. We could not execute springdoc-openapi for project OCVN to evaluate its description of the `@ModelAttribute` annotation, and Prophet wrongly reported the model attribute object as a single parameter.

3.3.2. JAX-RS's FormParam annotation

Project Features-service uses the `@FormParam` annotation, which expects receiving the annotated parameters in an URL-encoded string request body (Eclipse Foundation, 2026a). Swagger Core ignored this annotation and the corresponding parameters when generating OpenAPI descriptions. Respector, the second approach supporting JAX-RS, correctly reported the corresponding request bodies.

3.3.3. Jackson's JsonProperty annotation

Three projects use Jackson's `@JsonProperty` annotation, which translates Java field or getter method names to the transmitted JSON property names, i.e., the field names of the API data model. Respector and Prophet missed such annotations in the Spring Boot project Ur-Codebin. Only springdoc-openapi handled the annotation correctly and translated all property names. For the JAX-RS projects Cassandra and Kafka, Respector also missed the `@JsonProperty` annotations. Instead, it again reported the internal Java field names as property names. Swagger Core correctly translated the property names according to their annotations for both JAX-RS projects.

3.3.4. Request body detection

Respector missed one Spring `@RequestBody` annotation in project CatWatch and four in project ProxyPrint. We could not explain why Respector missed these five instances because we could not observe any differences from other request body annotation occurrences. Prophet also missed the same annotation in project CatWatch, whereas it detected the four annotations in project ProxyPrint. Only springdoc-openapi correctly reported the annotated parameter in project CatWatch. However it could not generate an OpenAPI description for project ProxyPrint.

Similarly, Respector missed arbitrary request body parameters in the JAX-RS projects Cassandra, Kafka, and Senzing but detected others. Swagger Core identified most request body parameters in project Cassandra but also missed many in project Kafka. Note that JAX-RS does not provide a request body annotation. Instead, the only unannotated handler parameter automatically marks the request body parameter.

3.3.5. Getter methods in abstract classes and interfaces

Spring Boot and JAX-RS derive API data models from Java classes and interfaces by mapping their fields and getter methods to JSON properties. Respector missed fields derived from getter methods declared in interfaces or abstract classes when reporting request bodies. This occurred in the projects Digdag, EnviroCar, and Kafka.

3.3.6. Project-specific implementation details

Finally, we found two reasons for incorrect parameter detection which we attribute to specific implementation details. Project EnviroCar manually implements paging logic that is hidden behind interfaces and uses runtime injection to obtain the request's paging parameters, and project Senzing overloads API operations that differ in the set of parameters based on their consumed and produced HTTP media types. Respector, the only approach that could analyze these two projects, did not detect the corresponding parameters.

3.4. API response codes

We reiterate that Prophet did not report any response codes and we therefore applied the OAS *default* response code during conversion. Similarly, Swagger Core did not detect response codes from the source code, but only from manually annotated documentation, such as Swagger's `ApiResponse` annotation. If a handler did not contain such annotations, Swagger Core reported the *default* response code without any additional information.

For the other two approaches, Respector and springdoc-openapi, we found the following reasons for misidentified responses.

3.4.1. Built-in exception handling

None of the approaches fully considered the frameworks' built-in exception handling logic that translates Java exceptions to response codes (cf. Section 2.1.3). For instance, four handlers in the Spring Boot project CatWatch throw an `UnsupportedOperationException` which translates to the response code *403 Forbidden*. Respector reported such responses as *500 Internal Server Error*. springdoc-openapi identified and correctly translated exceptions only if they were thrown directly

inside the handler, and ignored exceptions thrown in handler callees. Both approaches missed or misclassified exceptions for the Spring Boot projects CatWatch, Ohsome, and Quartz.

For the JAX-RS projects Kafka and Senzing, Respector could not translate the framework's Java exceptions, e.g., `BadRequestException`, to their corresponding response codes and instead reported *500 Internal Server Error*.

3.4.2. Custom exception handling

Multiple projects implement custom exception handling, such as registering exception handlers in a configuration class or manually translating exceptions to status codes. For instance, the Spring Boot project Ur-Codebin manually translates exceptions to response codes in `@ExceptionHandler`-annotated methods, and the JAX-RS project Digdag registers multiple `GenericJsonExceptionHandlers` with the exception class and status code as constructor parameters.

Respector did not report the correct response codes from such custom logic in the projects CWA (internal), Ur-Codebin, Digdag, EnviroCar, Gravitee v4, and Kafka, and instead reported the default error response *500 Internal Server Error*. Similarly, springdoc-openapi did not detect such custom exception handling logic and reported *500*.

3.4.3. Default success response codes

Respector misclassified the success response code of void handlers and handlers returning null as *204 No Content* in the projects OCVN, Ohsome, ProxyPrint, and Kafka, and incorrectly reported *206 Partial Content* in project EnviroCar (cf. Section 2.1.3). springdoc-openapi did not suffer from this shortcoming, but could not execute for two of the three affected Spring Boot projects.

3.4.4. Built-in response code methods

Respector did not detect multiple built-in methods for setting the response code that are used in the JAX-RS projects Cassandra and EnviroCar, e.g., `Response::created` or `Response::serverError`. Instead, it reported the default response code *200 OK* if no other response codes were found.

3.4.5. Propagated exceptions

Out of the existing approaches, only Respector analyzed callees of the original handler method. However, many handlers in the projects EnviroCar and Senzing catch their callees' exceptions and rethrow them or throw another one, which is not detected by Respector. springdoc-openapi did not detect exceptions thrown in handler callees in the Spring Boot projects CatWatch and Quartz, independently if they were called inside try-catch blocks or not.

3.4.6. Project-specific implementation details

Again, we found project-specific implementation details that were not analyzed correctly. The JAX-RS projects Cassandra and Digdag use lambda expressions that encapsulate the handler logic to catch and translate potential exceptions. Similarly, the Spring Boot project CWA (external) uses lambda expressions for asynchronous computations that eventually return a response. Respector did not detect status codes created inside of lambda functions and instead reported the default response code *200 OK*. springdoc-openapi correctly identified such responses for the Spring Boot project CWA (external).

Project Kafka uses project-specific asynchronous response objects that contain various status codes, which Respector did not identify as such. Instead, it reported *204 No Content*.

3.5. Data model representation accuracy

We reiterate that the GT+ document represents all parameter object fields as individual parameters to evaluate the completeness of the object schemas generated by the approaches (cf. Section 2.1.2). Furthermore, the GT+ document contains approximate response data types, e.g., primitive types, arrays, or objects, but not the detailed schemas of response objects (cf. Section 2.1.3). Hence, as part of this evaluation, we manually assessed the data model representation accuracy of generated OpenAPI descriptions based on the source code. We define an accurate representation as correctly describing the data models in the *#/components/schemas* section and preserving object inheritance information.

We did not consider Prophet for this comparison because it generates a custom JSON format instead of OpenAPI descriptions and only reports the string literals in Java method signatures as parameter and response types (cf.Section 3.1.2).

3.5.1. Usage of OpenAPI schemas section

After investigating Respector's OpenAPI descriptions, we found that it generated anonymous schemas containing all data model information directly inside the corresponding parameter or response section of the OpenAPI description. It did not generate any explicitly named schemas in the *#/components/schemas* section and thereby discarded the semantic information and relationships of data models.

Contrary, springdoc-openapi and Swagger Core placed data model schemas in the *#/components/schemas* section of the OpenAPI descriptions and referenced them in the corresponding parameter and response sections.

3.5.2. Map data structure

The projects CatWatch, OCVN, ProxyPrint, Cassandra, Digdag, EnviroCar, Gravitee v4, Kafka, and Senzing use the map data structure, also known as dictionary, in parameters and responses. The OAS describes maps as objects. It implicitly defines the map's key type as string and provides the `additionalProperties` keyword for the map's value type (OpenAPI Initiative, 2026c).

We found that Respector did not describe the map value type but only reported the data structure as a regular object without any additional information. In contrast, springdoc-openapi and Swagger Core correctly reported the value type by using the `additionalProperties` keyword.

3.5.3. Object inheritance

All projects use objects as API parameters and responses, and all projects use objects with fields that reference other objects. For instance, project EnviroCar returns `Groups` where the `owner` field is of object type `User`. In addition to such regular objects, the projects ProxyPrint, Quartz, EnviroCar, and Gravitee v4 use derived objects, i.e., objects inheriting from a superclass, as parameters, and the projects CWA, OCVN, ProxyPrint, Quartz, EnviroCar, Gravitee v4, RESTcountries, and Senzing return derived objects as responses.

For derived objects used as parameters, Respector flattened all inherited fields into one schema and thereby discarded the inheritance information. Moreover, Respector did not consider the inheritance information of derived objects used in responses at all. For instance, an operation in project CWA returns a data model named `InternalTestResult` with multiple inherited fields. Respector only reported a single field that is declared directly inside the class without considering the inherited fields. Surprisingly, we discovered that Respector did not generate any meaningful object schema information for some projects. For the above example of `Groups` in project EnviroCar, Respector only reported the two responses shown in Listing 3. One is a success response (Line 2) returning a generic OAS object (Line 7) with the fully qualified class name as object title (Line 8), and the second is a *default* response without any information (Line 10).

```
"responses": {
  "200": {
    "description": "OK",
    "content": {
      "application/json": {
        "schema": {
          "type": "object",
          "title": "org.envirocar.server.core.entities.
              Group"
  } } } },
  "default": {
    "description": "others"
} }
```

Listing 3: Two responses of the operation `GET /groups/{group}` in project EnviroCar with the data model generated by Respector.

springdoc-openapi flattened all inherited fields of parameter and response objects into a single schema and, similar to Respector, discarded all inheritance information. Swagger Core reported inheritance information using the OAS `allOf` keyword only when the inheritance relation was specified with corresponding Swagger documentation annotations.

Answer to RQ1: Shortcomings of Existing Approaches

We identified several shortcomings of existing approaches that impact the quality of OpenAPI descriptions extracted from Java web service projects.
The static analysis approaches ignored Spring profiles, which impact the service runtime behavior, and incorrectly reported path constraints. All approaches ignored or incorrectly handled several framework annotations that define request parameters and translate internal Java field names to JSON property names. They also failed to detect many built-in and custom exception handling patterns and incorrectly reported multiple response codes.
Moreover, the existing approaches have failed to generate explicit schema information for parameter and response data models in the corresponding OpenAPI section, discarded inheritance information and inherited fields, or even omitted basic object schemas.
Finally, the dynamic analysis approaches did not execute for 5 out of 15 projects due to missing or invalid run configurations.
These shortcomings hamper the applicability of existing approaches to Java projects and motivate a new, improved approach, which we introduce in the next section.

4. AutoOAS approach

Based on the identified shortcomings of existing approaches, we propose our approach named AutoOAS for generating accurate OpenAPI descriptions from Java source code. AutoOAS uses Spoon (Pawlak et al., 2015) to parse and statically analyze the Java source code, its inheritance hierarchy, and annotations. It does not require bytecode or a running, accessible service, which simplifies the OpenAPI generation process by omitting compilation, execution, and runtime dependency management. Currently, AutoOAS supports two popular frameworks for developing Java web services, namely Spring Boot and JAX-RS. We list the supported framework annotations in Table 6.

Fig. 1 shows the workflow of AutoOAS. AutoOAS identifies all Java classes containing REST API operation definitions and groups them based on their assignment to runtime profiles. It generates one OpenAPI description for each profile. Notably, JAX-RS does not provide a mechanism comparable to Spring profiles, and AutoOAS generates only a single OpenAPI description for this framework. AutoOAS first identifies the operations, parameters, and responses from the source code and uses the information to generate the OAS *#/paths* section. Then, it detects the data models referenced as parameter and response objects and generates the corresponding schema information in the OAS *#/components/schemas* section. Notably, if a Spring Boot service configuration defines multiple active profiles at once, merging the individual OpenAPI descriptions provides the same analysis results as designing AutoOAS to consider multiple profiles simultaneously. The OpenAPI descriptions generated by AutoOAS are mergeable with set unions of their *#/paths* sections, which are disjoint, and their *#/components/schemas* sections, which are either disjoint or contain identical duplicates. This is because all operations must be unique according to the OAS, and all data model classes are uniquely identifiable at compile time.

In the following, we present the three stages of AutoOAS in detail, maintaining the distinction between *operation* and *handler* (cf. Section 1).

4.1. Source code parsing

AutoOAS parses the project's source code with Spoon to identify all Java classes that contain operation definitions. For this, AutoOAS detects annotations that mark the parsed Java classes as *controllers*, such as Spring Boot's `@RestController` and JAX-RS's `@Path`. AutoOAS automatically recognizes the framework based on these annotations. If a project does not contain any of the supported controller annotations, AutoOAS will exit with a message that the project's framework is not supported. AutoOAS also scans the project for a JAX-RS `Application` subclass that is annotated with `@ApplicationPath`. If the annotation defines a global path prefix for all API operations, AutoOAS extracts and reports the corresponding prefix in the OAS *#/servers* section.

Next, AutoOAS assigns each controller class to its assigned runtime profile set(s) or to all profile sets, including the default profile, if they are not explicitly assigned to at least one. Then, AutoOAS generates one individual OpenAPI description for each profile as described in the following sections.

4.2. Analyzing API operations, parameters, and responses

In this stage, AutoOAS generates the *#/paths* section of the OpenAPI description by analyzing all operation definitions in the controller classes of a single runtime profile.

4.2.1. API operations

AutoOAS considers superclasses of controllers because they potentially contain additional configurations or operations. AutoOAS also detects Java methods that are annotated with `@Path` and return a controller class to support JAX-RS sub-resource locators. It adds all sub-resources to the set of controllers under analysis.

Starting from the controller classes, AutoOAS traverses each controller's inheritance hierarchy and identifies all Java methods, i.e., handlers, annotated with `@RequestMapping` or specialized HTTP method mapping annotations, e.g., `@GetMapping` and `@GET`, as operation definitions. AutoOAS extracts the API operation's path and HTTP methods directly from the handler's mapping annotations.

4.2.2. API parameters

Next, AutoOAS analyzes the parameters. It identifies each parameter name and type from the handler's signature. If a parameter annotation explicitly defines the parameter name, e.g., `@RequestParam(''param_name'')`, AutoOAS renames it accordingly. It also derives the parameter location, i.e., path, query, or header, from the annotation. If a path parameter contains a regular expression (regex) constraint following the syntax `/{parameter:regex}`, AutoOAS removes the regex from the path and adds it to the OAS `pattern`

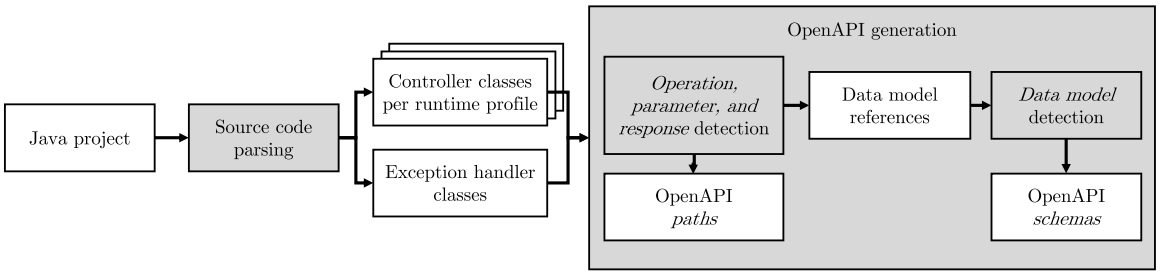


Fig. 1. The AutoOAS workflow for generating OpenAPI descriptions from source code.

Table 6
Spring Boot and JAX-RS annotations supported by AutoOAS.

Analysis Step	Spring Boot	JAX-RS
Controller classes	@Controller, @RestController, @RepositoryRestController	@Path
Runtime profiles	@Profile	–
API operations	@PostMapping, @PutMapping, @PatchMapping, @GetMapping, @DeleteMapping, @RequestMapping	@POST, @PUT, @PATCH, @GET, @DELETE
API parameters	@PathVariable, @RequestParam, @RequestHeader, @RequestBody, @ModelAttribute	@PathParam, @QueryParam, @HeaderParam, @FormParam
API responses	@ResponseStatus	@Produces
Exception handling	@ControllerAdvice, @RestControllerAdvice, @ExceptionHandler	@Provider
Bean Validation	@NotNull, @NotEmpty, @Pattern, @Size, @DecimalMin, @DecimalMax, @Min, @Max	
Data model translation	@JsonProperty	

constraint field of the parameter. AutoOAS converts parameter objects annotated with Spring Boot's `@ModelAttribute` into individual query parameters based on the object's fields. It also supports inherited fields by recursively iterating the object's inheritance hierarchy.

The Spring Boot `@RequestBody` annotation marks a special parameter transferred in the HTTP request body, and AutoOAS uses the dedicated OAS keyword `requestBody` to describe it. Similarly, AutoOAS reports the unannotated handler parameter in JAX-RS as the request body. Furthermore, AutoOAS converts parameters annotated with the JAX-RS `@FormParam` annotation into fields of a single request body with the *application/x-www-form-urlencoded* media type.

Notably, AutoOAS does not describe object parameter schemas inline but generates a reference to a named schema in the *#/components/schemas* section of the OpenAPI description by using the `$ref` keyword. This enables the reusability of the API's data models.

4.2.3. API responses

AutoOAS analyzes each handler's return type and its body for `return` and `throws` statements to identify the responses. Handlers may return regular Java objects or framework-specific response wrappers that wrap Java objects and potentially contain HTTP response codes, e.g., Spring Boot's `ResponseEntity<T>` and JAX-RS's `Response`. AutoOAS detects response codes returned by both frameworks' response wrappers from the body as well as defined with Spring Boot's `@ResponseStatus` annotation. We limit the analysis of call chains inside handler bodies to a depth of one extra class for performance reasons. We assume that creating response wrappers is a controller's responsibility and is therefore typically implemented close to the handler logic. In case that no particular response wrapper or explicit `@ResponseStatus` annotation could be identified, AutoOAS reports the default response status code *200 OK*. If the handler has a void return type, AutoOAS reports the framework's default response code, *200 OK* for Spring Boot and *204 No Content* for JAX-RS.

AutoOAS extracts the data model schema of successful responses from the handler's return type or response wrapper creation in the handler's body and references a named schema in the *#/components/schemas* section. In addition to the `application/json` media type, AutoOAS detects annotations that explicitly declare a response body media type, such as JAX-RS's `@Produces`, and accordingly reports the type in the OpenAPI description.

Handlers may throw exceptions with the `throws` statement. Both frameworks provide mechanisms to translate exceptions to HTTP client and server error codes (cf. Section 2.1.3). Spring Boot provides the `@ExceptionHandler` method annotation. The Java method marked with this annotation is either declared inside a controller to translate the exceptions of all handlers inside the same controller, i.e., local exception handling, or declared in a separate class annotated with `@ControllerAdvice` for translating the exceptions of any handler, i.e., global exception handling. JAX-RS provides multiple framework exceptions such as `BadRequestException` that are automatically translated, i.e., built-in exception handling, and the `ExceptionMapper` interface in combination with the `@Provider` class annotation for global exception handling.

AutoOAS detects such exception-handling behavior. For this, it tracks all classes defining global exception handlers during the parsing stage described in Section 4.1. When encountering a `throws` statement in a handler's or handler callee's body, AutoOAS translates the thrown exception to a response code. It first attempts to match the exception against built-in exceptions or to identify a local exception handler, and resolves the corresponding response code. If it could not translate the exception, it searches for a match among the global exception handlers to determine the corresponding response code. Unresolved exceptions result in an *500 Internal Server Error* and are reported accordingly.

4.3. API data models

In this stage, AutoOAS generates the data models used as parameters and responses. We differentiate between two types of data models: simple types and named schemas. According to the OAS (OpenAPI Initiative, 2026a), AutoOAS reports simple types directly inside the corresponding parameter or response description and named schemas in the *#/components/schemas* section of the OpenAPI description.

4.3.1. Simple types

AutoOAS supports Java's built-in primitive and reference types, e.g., int or String. It maps primitive types and their wrapper objects to the basic types and formats of the OAS (OpenAPI Initiative, 2026b), e.g., int to *integer/int32*, double to *number/double*. It converts strings to the OAS *string* type, and explicitly lists Java enum constants in the OAS *enum* field. It represents Java arrays and lists as the OAS *array* type, and Java sets as the OAS *array* type with the `uniqueItems` flag set to true. Finally, it reports the map data structure as an *object* and uses the `additionalProperties` keyword to describe the value type.

AutoOAS supports Bean Validation constraint annotations, e.g., `@NotNull` for reference types or `@Min` and `@Max` for numeric types. It reports the corresponding OAS constraints for the type, e.g., `required`, `minimum`, and `maximum`.

4.3.2. Named schemas

AutoOAS reports custom Java classes used as data models in parameters and responses as named schemas in the OAS *#/components/schemas* section. For this, AutoOAS records all Java classes that were referenced when generating the parameters and responses during the previous stage. AutoOAS reports the named schemas by describing each field of a data model class as either a simple type or as another reference to a named schema, depending on its Java type. If a field is annotated with Jackson's `@JsonProperty`, AutoOAS translates the internal Java field name to the property name defined by the annotation. Additionally, AutoOAS marks required fields of the schema with the OAS `required` keyword. It considers fields as required if they contain the `@NotNull` or `@NotEmpty` Bean Validation annotations.

If the data model class is a derived class, i.e., a class inheriting fields from a superclass, AutoOAS uses the OAS `allOf` keyword to create a combined schema which contains the schema of the current class and a reference to its superclass. It then generates the named schema of the superclass and, if necessary, references its superclass again. Thereby, AutoOAS recursively scans the inheritance tree and retains transitive inheritance information in the OpenAPI description instead of creating a flat data model schema, i.e., describing all fields inside the same schema.

4.3.3. Wrapper classes and error handling

AutoOAS automatically handles framework-specific response wrappers, i.e., Spring's `ResponseEntity<T>` and JAX-RS's `Response`, as well as asynchronous return types, i.e., Spring's `DeferredResult<T>` and JAX-RS's `CompletionStage<T>`. AutoOAS extracts and reports the type parameter `T` or, in its absence, analyzes the creation of the response wrapper in the handler body. Similarly, AutoOAS handles the Java `Optional<T>` wrapper by reporting the wrapped type parameter `T`.

If AutoOAS cannot detect a data model type, it uses a special named schema called *UNSPECIFIED_TYPE*. If it detects a type but cannot resolve it, e.g., because the defining class is located in an inaccessible source package, AutoOAS creates a named schema from the class name and reports the package name in the OAS `externalDocs` field.

5. Evaluation of API operation, parameter, and response code detection

In this section, we answer RQ2 by evaluating the detection capabilities of AutoOAS with the GT+ document described in Section 2. We compare the precision and recall of AutoOAS, Respector, Prophet, springdoc-openapi, and Swagger Core for identifying API operations, parameters, and response codes. To generate the OpenAPI descriptions of Respector, Prophet, springdoc-openapi, and Swagger Core, we reused the setup from Section 3.1.

5.1. Evaluation procedure

We created a script to automatically calculate precision and recall. For each approach, it first transforms the generated OpenAPI descriptions into lists of operations, parameters, and response codes per project. For evaluating the parameters, it flattens the named schemas of AutoOAS, also considering the inheritance hierarchy, to compare them to the individual parameters of the GT+ document (cf. Section 2.1.2). Similarly, it flattens the request body objects of Respector, springdoc-openapi, and Swagger Core.

Next, it compares the list entries to the operations, parameters, and response codes in the GT+ document. An identified operation is a true positive (TP) if the combination of endpoint path and HTTP method exists in the GT+ document, an identified parameter or object parameter field is a TP if the operation and parameter or field name match with the GT+ document, and an identified response is a TP if the operation and response code match. The script also counts false positives (FP), i.e., identified operations, parameters, and responses not existing in the GT+, and false negatives (FN), i.e., not identified operations, parameters, or responses that do exist in the GT+. With these results, it calculates precision and recall as follows:

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}.$$

We define the precision without any TP or FP findings as 0% to penalize approaches not reporting any operations, parameters, or response codes. We calculated the overall precision and recall as the weighted average of the individual results, and we highlight that we only considered successful analyses for the overall scores. In particular, this favors the runtime approaches, springdoc-openapi and Swagger Core, where we could not analyze two Spring Boot and three JAX-RS projects, respectively. However, we argue that, given sufficient domain knowledge, these approaches could generate OpenAPI descriptions with evaluation results similar to those of the successful analyses. The precision and recall results for each project and each approach are presented in Table 7. In the following, we describe the results in detail.

5.2. API operations

Only AutoOAS correctly identified all operations. It obtained an overall precision and recall of 100%. AutoOAS generated two OpenAPI descriptions with the correct subsets of operations for the two Spring profiles, *external* and *internal*, in project CWA, and it detected the `@ApplicationPath` annotation in project RESTcountries with the correct path prefix. Furthermore, it correctly captured the regular expression constraints of parameters defined in the paths of project OCVN and removed them from the path string.

Respector incorrectly reported the operations of two projects. It created a single OpenAPI description for both Spring profiles of project CWA and missed the path prefix in project RESTcountries. However, project CWA contains only three operations per runtime profile and, consequently, did not have a noticeable impact on Respector's overall results of 97% precision and recall.

Swagger Core correctly reported all operations of the projects we could analyze, resulting in an overall precision and recall of 100% for JAX-RS projects.

springdoc-openapi achieved an overall precision of 99% and a recall of 95% for the analyzed Spring Boot projects. It missed multiple operations in project CatWatch, but we could not identify any differences to the detected operations in the project's source code. Contrary, the missed handlers contain manually annotated Swagger documentation, same as the detected handlers. Next, springdoc-openapi did not report several operations in project Quartz that are annotated as hidden from documentation. However, such Swagger annotations do not change the runtime behavior of the API and the corresponding operations are regularly accessible.

Table 7
Precision and **R**ecall results of the five approaches. The GT+ column contains the number (#) of API operations, parameters, and responses for each project. Results are highlighted in **bold** when an approach achieved both highest precision and highest recall for all three categories. Unsuccessful analyses are marked with an 'x', and unsupported frameworks are marked with a '–'.[1] Overall averages only successful analyses.

Project		GT+	AutoOAS		Respector		Prophet		springdoc		Swagger Core	
		#	P	R	P	R	P	R	P	R	P	R
Catwatch	operations	14	**1.00**	**1.00**	1.00	1.00	0.60	0.21	1.00	0.57	–	–
	parameters	36	**1.00**	**1.00**	1.00	0.97	0.00	0.00	1.00	0.33	–	–
	responses	19	**1.00**	**0.79**	0.63	0.63	0.00	0.00	0.62	0.42	–	–
CWA (external)	operations	3	1.00	1.00	0.60	1.00	0.00	0.00	1.00	1.00	–	–
	parameters	10	1.00	1.00	0.69	0.90	0.00	0.00	1.00	1.00	–	–
	responses	7	0.71	0.71	0.11	0.14	0.00	0.00	0.70	1.00	–	–
CWA (internal)	operations	3	**1.00**	**1.00**	0.60	1.00	0.00	0.00	1.00	1.00	–	–
	parameters	6	**1.00**	**1.00**	0.46	1.00	0.00	0.00	0.83	0.83	–	–
	responses	9	**0.88**	**0.78**	0.33	0.33	0.00	0.00	0.58	0.78	–	–
OCVN	operations	278	**1.00**	**1.00**	1.00	1.00	0.95	0.45	x	x	–	–
	parameters	5,018	**1.00**	**0.94**	0.16	0.16	0.06	0.00	x	x	–	–
	responses	278	**1.00**	**1.00**	0.81	0.89	0.00	0.00	x	x	–	–
Ohsome	operations	159	1.00	1.00	1.00	1.00	1.00	0.50	1.00	1.00	–	–
	parameters	1,937	0.00	0.00	1.00	0.99	0.00	0.00	0.00	0.00	–	–
	responses	179	1.00	0.89	0.37	0.79	0.00	0.00	1.00	0.89	–	–
ProxyPrint	operations	75	1.00	1.00	1.00	1.00	0.99	0.89	x	x	–	–
	parameters	157	1.00	0.78	0.99	0.92	0.30	0.21	x	x	–	–
	responses	103	1.00	0.98	0.97	0.95	0.00	0.00	x	x	–	–
Quartz	operations	14	1.00	1.00	**1.00**	**1.00**	0.07	0.07	0.91	0.71	–	–
	parameters	15	1.00	1.00	**1.00**	**1.00**	0.00	0.00	0.43	0.20	–	–
	responses	20	1.00	0.75	**1.00**	**0.80**	0.00	0.00	0.45	0.50	–	–
Ur-Codebin	operations	6	1.00	1.00	1.00	1.00	1.00	1.00	**1.00**	**1.00**	–	–
	parameters	14	1.00	1.00	0.43	0.43	0.13	0.07	**1.00**	**1.00**	–	–
	responses	15	0.55	0.40	0.45	0.33	0.00	0.00	**1.00**	**0.40**	–	–
Overall[1] Spring Boot	operations	552	**1.00**	**1.00**	1.00	1.00	0.93	0.51	0.99	0.95	–	–
	parameters	7,193	**0.73**	**0.68**	0.41	0.41	0.05	0.00	0.04	0.02	–	–
	responses	630	**0.98**	**0.93**	0.69	0.83	0.00	0.00	0.90	0.79	–	–
Cassandra	operations	50	**1.00**	**1.00**	1.00	1.00	–	–	–	–	1.00	1.00
	parameters	95	**1.00**	**1.00**	0.59	0.58	–	–	–	–	1.00	0.97
	responses	123	**1.00**	**0.98**	0.89	0.41	–	–	–	–	1.00	0.65
Digdag	operations	41	1.00	1.00	1.00	1.00	–	–	–	–	x	x
	parameters	97	0.99	0.99	1.00	0.82	–	–	–	–	x	x
	responses	87	0.88	0.77	0.98	0.51	–	–	–	–	x	x
EnviroCar	operations	128	**1.00**	**1.00**	1.00	1.00	–	–	–	–	x	x
	parameters	449	**1.00**	**0.88**	1.00	0.45	–	–	–	–	x	x
	responses	507	**1.00**	**0.77**	0.78	0.27	–	–	–	–	x	x
Features-service	operations	18	1.00	1.00	**1.00**	**1.00**	–	–	–	–	1.00	1.00
	parameters	35	1.00	1.00	**1.00**	**1.00**	–	–	–	–	1.00	0.83
	responses	29	1.00	0.72	**1.00**	**1.00**	–	–	–	–	0.00	0.00
Gravitee v4	operations	28	1.00	1.00	1.00	1.00	–	–	–	–	1.00	1.00
	parameters	110	1.00	0.85	0.99	0.97	–	–	–	–	0.93	0.97
	responses	47	0.94	0.94	0.86	0.81	–	–	–	–	0.00	0.00
Kafka	operations	69	1.00	1.00	1.00	1.00	–	–	–	–	1.00	1.00
	parameters	193	0.98	0.96	0.97	0.84	–	–	–	–	0.99	0.88
	responses	124	0.80	0.66	0.35	0.23	–	–	–	–	0.00	0.00
RESTcountries	operations	27	**1.00**	**1.00**	0.00	0.00	–	–	–	–	1.00	1.00
	parameters	37	**1.00**	**1.00**	0.00	0.00	–	–	–	–	1.00	0.95
	responses	72	**1.00**	**1.00**	0.00	0.00	–	–	–	–	0.00	0.00
Senzing	operations	34	**1.00**	**1.00**	1.00	1.00	–	–	–	–	x	x
	parameters	164	**1.00**	**0.99**	1.00	0.94	–	–	–	–	x	x
	responses	103	**1.00**	**0.94**	1.00	0.48	–	–	–	–	x	x
Overall[1] JAX-RS	operations	395	**1.00**	**1.00**	0.93	0.93	–	–	–	–	1.00	1.00
	parameters	1,180	**1.00**	**0.93**	0.93	0.67	–	–	–	–	0.98	0.92
	responses	1,092	**0.97**	**0.82**	0.74	0.34	–	–	–	–	0.31	0.20
Overall	operations	947	**1.00**	**1.00**	0.97	0.97	–	–	–	–	–	–
	parameters	8,373	**0.77**	**0.72**	0.48	0.45	–	–	–	–	–	–
	responses	1,722	**0.97**	**0.86**	0.72	0.52	–	–	–	–	–	–

Prophet obtained an overall precision of 93% and a recall of 51%. The results are only comparable to AutoOAS and Respector for two projects, Ur-Codebin and ProxyPrint. Prophet did not detect any operations of project CWA correctly because it could not resolve the constants in the endpoint paths. We observed the same shortcoming in project Quartz.

5.3. API parameters

AutoOAS correctly identified and converted the `@ModelAttribute` annotations in project OCVN. Furthermore, it detected `@JsonProperty` annotations on Java fields and getter methods and correctly reported the translated API field names. In contrast to existing approaches, it deterministically detected all parameters annotated with `@RequestBody` in the projects CatWatch and ProxyPrint.

AutoOAS obtained an overall precision of 77% and recall of 72% for detecting API parameters across all projects, compared to Respector's 48% and 45% for precision and recall. It also outperformed Prophet and springdoc-openapi for detecting parameters in Spring Boot projects, and Swagger Core for JAX-RS projects. Prophet detected many parameters with simple types, but could not translate parameter names based on annotations and did not report parameter object fields, resulting in low precision and recall. For the projects that springdoc-openapi and

Swagger Core analyzed, they achieved precision and recall results that were less than or equal to those of AutoOAS. The only exception is project Gravitee v4, where Swagger Core reported many parameters resulting in higher recall but lower precision than AutoOAS. Still, AutoOAS missed several parameters in seven projects and incorrectly reported parameters in two projects due to the following reasons.

5.3.1. HTTP servlet objects

Two Spring Boot projects use raw HTTP request and response objects that encapsulate arbitrary parameters at runtime. Ohsome exclusively uses `HttpServletRequest` for all operations, and ProxyPrint uses `HttpServletRequest` as well as `WebRequest`. These objects contain dynamic key–value pairs of parameter names and values, and are passed to handler callees for processing.

AutoOAS could not detect the actual parameters encapsulated in such objects without fully analyzing the source code control and data flow. We decided to accept this limitation because Spring Boot recommends request parameter handling using the annotation-based programming model (Spring, 2026a), as done by the other projects and supported by AutoOAS. Similarly, springdoc-openapi ignored `HttpServletRequest` objects in project Ohsome per default (springdoc, 2026b). Hence, AutoOAS and springdoc-openapi did not report any parameters in project Ohsome, resulting in 0% precision and recall. For project ProxyPrint, AutoOAS missed the encapsulated parameters, and springdoc-openapi could not generate the OpenAPI description. Prophet naively reported all HTTP request and response objects as the API parameters, resulting in 0% precision and recall for project Ohsome and low precision and recall for project ProxyPrint. Respector was the only approach that detected encapsulated parameters. Its symbolic analysis traced the servlet objects' data flow to the method calls that access concrete parameters by name. Correspondingly, Respector achieved 100% precision and 99% recall for the API parameters in project Ohsome, and slightly lower results for ProxyPrint.

For JAX-RS, we found that project Gravitee v4 uses multiple raw `HttpHeaders` objects to receive header parameters. `HttpHeaders` objects again encapsulate arbitrary parameters at runtime and none of the approaches detected these parameters.

5.3.2. Advanced Jackson annotations

AutoOAS was the only static analysis approach that supported the `@JsonProperty` annotation, which is used in three projects. However, it missed the Jackson annotations `@JsonDeserialize`, used in project Digdag, and `@JsonValue`, used in project Kafka. We consider these instances to be negligible, considering the total number of parameters, and leave the handling of the two annotations to future work. By ignoring these annotations, AutoOAS reported incorrect parameter names, resulting in 99% precision for project Digdag and 98% precision for project Kafka. Note that this is the only reason for AutoOAS to report false positive parameters.

5.3.3. Class-level parameter annotations

Project Gravitee v4 binds multiple path parameters to instance fields of the controller class using class-level parameter annotations. AutoOAS does not detect such class-level bindings, explaining the 85% recall for Gravitee v4. Respector and Swagger Core correctly reported these parameters, resulting in higher recall than AutoOAS. However, Gravitee v4 is the only project using this mechanism.

5.3.4. Project-specific implementation details

We identified multiple reasons for undetected parameters, which we attribute to specific implementation details.

Project EnviroCar manually implements parameter resolution for paging that is hidden behind interfaces and not detected by existing approaches, as explained in Section 3.3.6. Project Kafka uses a custom wrapper class for lazy deserialization of object parameters, which are also not reported by existing approaches. Similarly, we decided against tailoring AutoOAS to such project-specific implementations to avoid overfitting our approach to the evaluation dataset.

Project Kafka contains a single *GET* method operation that expects a request body parameter. However, such parameters for *GET* methods have undefined semantics in REST and are not supported by the OAS. Respector reported this parameter as a query parameter, which would result in incorrect REST API calls. Instead, we chose to discard such parameters in reports generated by AutoOAS.

Finally, project Senzing overloads two API operations that differ based on the media types they consume and produce. AutoOAS analyzed only one handler per operation and correspondingly missed two parameters for this project.

5.4. API response codes

AutoOAS improved the detection of response codes by correctly returning *200 OK* for Spring Boot void handlers and handlers returning null, supporting built-in response code methods, and considering exception handling functionality of both frameworks. It achieved an overall precision of 97% and recall of 86% for identifying response codes, compared to Respector's 72% precision and 52% recall across all projects. springdoc-openapi achieved comparable results for the projects it could analyze. Prophet did not report response codes from the source code, and Swagger Core only reported manually annotated response code documentation in project Cassandra.

Still, AutoOAS missed response codes in 13 projects and incorrectly reported response codes in five projects.

5.4.1. Custom exception handling

The various implementations for returning error response codes are the main reason for incorrect detection of API responses. Misclassifications result in both, FP findings of the default error response code *500 Internal Server Error* and FN findings of the actual error response code, thereby reducing both precision and recall.

In project Ur-Codebin, AutoOAS and Respector reported *500 Internal Server Error* instead of manually translated response codes *400 Bad Request*, *404 Not Found*, and *409 Conflict*. springdoc-openapi did not detect any error response codes for this project, and hence, created FN but no FP findings. This resulted in a comparable recall but higher precision than AutoOAS and Respector. Notably, Ur-Codebin is the only Spring Boot project that does not use the `@ResponseStatus` annotation.

The JAX-RS project Digdag manually translates the `IllegalArgumentException` to a *400 Bad Request* response code, which again resulted in the FP response code *500*. Furthermore, the JAX-RS projects Digdag and Kafka implement and throw custom exceptions containing the error response code as an instance field, which was not detected by any approach.

5.4.2. Propagated exceptions

AutoOAS did not detect propagated exceptions from handler callees in the projects CatWatch, Quartz, Features-service, and Kafka. The `throws` statements in the callee chain were outside of the analysis scope of AutoOAS and therefore not considered. Project Senzing conditionally builds and returns exceptions inside callees, which are then thrown by the caller. In such cases, AutoOAS only detected the common supertype of the callee's exceptions and not individual types. Respector's symbolic execution identified many propagated exceptions but failed to translate them to the correct response codes.

AutoOAS also missed propagated exceptions that were hidden inside lambda functions in projects CWA (internal), Cassandra, and Kafka. Of the existing approaches, only springdoc-openapi correctly reported the exceptions thrown inside lambda functions in the Spring Boot project CWA (internal).

5.4.3. Propagated response codes

Similar to propagated exceptions, AutoOAS did not detect `ResponseEntity` objects containing *201 Created* that are returned from handler callees in both profiles of project CWA.

Contrary, AutoOAS detected propagated responses in project Gravitee v4 and missed that these responses are not further returned by the handler but instead used by the business logic. The corresponding FP findings explain the 94% precision for Gravitee v4.

5.4.4. Project-specific implementation details

Similarly to project Senzing, project EnviroCar overloads an API operation that differs based on its consumed and produced media types. AutoOAS analyzed only one overload handler, which resulted in two FN responses. However, we consider this form of API operation overloading to be untypical in practice.

As described in Section 3.4.6, project Kafka uses project-specific asynchronous response objects that contain various response codes. AutoOAS did not detect this peculiarity and instead reported the default response code *200 OK*.

Answer to RQ2: Detection Accuracy of AutoOAS

AutoOAS achieved the highest overall precision and recall when identifying REST API operations, parameters, and responses. It outperformed the second-best approach, Respector, by 29 percentage points in precision and 27 in recall when identifying parameters, and by 25 percentage points in precision and 34 in recall when identifying response codes across all projects.

6. Evaluation of OpenAPI representation accuracy

We reiterate that the GT+ document lists all API operations, parameters, and response codes for all projects, but it does not capture OpenAPI-specific constructs such as parameter locations, request and response body media types, named schemas, or required flags. Hence, in this section, we evaluate the representation accuracy of the OpenAPI descriptions generated by AutoOAS to answer RQ3. For this, we created an evaluation script that compares the operations, parameters, request bodies, responses, and referenced data models of the generated OpenAPI descriptions against the GT+ OpenAPI descriptions described in Section 2. The script is available in our replication package (Lercher et al., 2026).

6.1. OpenAPI paths section constructs

First, we evaluated the representation accuracy of generated OpenAPI *#/paths* section constructs.

6.1.1. Evaluation procedure for paths constructs

We classified a generated *paths* section construct as a TP if it matched the corresponding GT+ element according to the following criteria: an API operation must have the same endpoint path and HTTP method; a parameter must have the same name, location, schema, and required flag; a request body must be present with the same required flag; a request body media type must be identical; a response must have the same HTTP status code; a response header must have the same name and schema; and a response body media type must be identical. Additionally, we counted FP and FN constructs if they only existed in the generated or the GT+ OpenAPI description, respectively.

For parameters and response headers, we considered schemas as equal if they have the same simple type or both reference a named schema in the OpenAPI *#/components/schemas* section. Consequently, we penalized approaches that did not represent complex schemas as named schemas, even though they may have generated structurally correct inline schemas. However, path, query, and header parameters as well as response headers typically have simple types, whereas complex data models are more commonly transmitted as JSON objects in request and response bodies. We separately evaluate the approaches' representation of such *schemas* constructs, including nested structures and schema references, in the next Section 6.2.

We report the precision and recall results for *paths* constructs of each approach in Table 8 and discuss them in detail below.

6.1.2. API parameters

We evaluated the detection of simple parameters and request body fields in Section 5.3. Regarding their representation accuracy in the OpenAPI descriptions, AutoOAS correctly reported the locations of simple parameters it detected, and it correctly referenced complex data models as named schemas in the *#/components/schemas* sections.

However, we found that AutoOAS ignored the `@DefaultValue` annotation of JAX-RS. The projects Kafka, Senzing, and Ur-Codebin use this annotation to define default values for simple parameters, which were not reported by AutoOAS or Respector. Only Swagger Core reported the default values of simple parameters in its OpenAPI descriptions.

Furthermore, we discovered that Respector did not correctly set the required flag for request bodies, which are required by default in both frameworks, Spring Boot and JAX-RS. AutoOAS, springdoc-openapi, and Swagger Core correctly reported this default behavior.

6.1.3. API responses

Section 5.4 evaluated the response code detection capabilities of the approaches. Accordingly, this evaluation focuses on the representation accuracy of response headers and bodies.

AutoOAS represented the largest number of response body media types correctly, with a precision of 97% and recall of 68% for Spring Boot projects, and 91% precision and 56% recall for JAX-RS projects. The second-best approach, Respector, achieved comparable results only for Spring Boot projects. We reiterate that Prophet did not generate any response constructs, and Swagger Core reported responses only from manually annotated documentation. However, we identified several reasons why AutoOAS produced incorrect response constructs.

First, AutoOAS and Respector did not detect any response headers that are returned by five projects: CWA (internal), Ohsome, EnviroCar, Features-service, and Gravitee v4. springdoc-openapi reported the single response header in project CWA (internal), but it missed all response headers in project Ohsome. Ohsome's undetected response headers are set in handler callees and are therefore not analyzed by springdoc-openapi.

Next, AutoOAS missed multiple response body media types. It did not detect nine `@Produces` annotations in the JAX-RS project Cassandra, resulting in FP and FN media types. However, AutoOAS detected other `@Produces` annotations in the same project and in other projects, and we could not identify any differences between the detected and undetected annotations in the project's source code. The Spring Boot project Ohsome uses `HttpServletResponse` servlet objects to return arbitrary API responses, where the response body media type is either `application/json` or `text/csv` depending on a query parameter. AutoOAS, Respector, and springdoc-openapi missed the `text/csv` response media type because it is only set along the handler's call chains. Notably, AutoOAS and springdoc-openapi are the only approaches that reported response body media types other than `application/json`.

Finally, the projects Gravitee v4 and Senzing use custom exception handling logic to append response bodies to error responses. Similar to the limitation in Section 5.4.1, AutoOAS and Respector could not identify such response bodies and only reported the response error codes.

Table 8
Overall **P**recision and **R**ecall of the five approaches compared to the GT+ OpenAPI descriptions per framework. Results are highlighted in **bold** when an approach achieved both highest precision and highest recall. Overall results average only successful analyses. Unsupported frameworks are marked with a '–'.

Framework		GT+	AutoOAS		Respector		Prophet		springdoc		Swagger Core	
		#	P	R	P	R	P	R	P	R	P	R
Spring Boot	Ops	62	**1.00**	**1.00**	**1.00**	**1.00**	0.74	0.39	1.00	0.86	–	–
	Params	274	0.54	0.52	**1.00**	**0.61**	0.00	0.00	0.21	0.11	–	–
	ReqBodies	13	1.00	0.85	0.00	0.00	0.00	0.00	**1.00**	**1.00**	–	–
	ReqBodyMT	13	1.00	0.92	1.00	0.85	0.38	0.23	**1.00**	**1.00**	–	–
	Resp	117	**0.92**	**0.65**	0.43	0.60	0.00	0.00	0.79	0.49	–	–
	RespHead	10	0.00	0.00	0.00	0.00	0.00	0.00	**0.10**	**0.10**	–	–
	RespBodyMT	71	**0.97**	**0.68**	0.96	0.65	0.00	0.00	0.77	0.57	–	–
JAX-RS	Ops	80	**1.00**	**1.00**	0.88	0.88	–	–	–	–	**1.00**	**1.00**
	Params	158	1.00	0.88	0.92	0.77	–	–	–	–	0.97	0.89
	ReqBodies	14	**0.93**	**0.93**	0.07	0.07	–	–	–	–	0.80	0.60
	ReqBodyMT	22	**1.00**	**0.64**	0.18	0.18	–	–	–	–	0.40	0.40
	Resp	214	**0.97**	**0.83**	0.44	0.38	–	–	–	–	0.22	0.13
	RespHead	12	0.00	0.00	0.00	0.00	–	–	–	–	0.00	0.00
	RespBodyMT	138	**0.91**	**0.56**	0.68	0.30	–	–	–	–	0.32	0.17

Ops=Operations, Params=Parameters, ReqBodies=Request bodies, ReqBodyMT=Request body media types, Resp=Responses, RespHead=Response headers, RespBodyMT=Response body media types.

6.2. OpenAPI schemas section constructs

After evaluating the OpenAPI *paths* section constructs, we assessed how accurately the approaches represented the schemas used in request and response bodies. While the previous evaluation considered whether they reported request and response body media types, it did not assess their representation of the corresponding schemas.

6.2.1. Evaluation procedure for schemas constructs

In the GT+ OpenAPI descriptions, we defined complex schemas as reusable named schemas in the OpenAPI *#/components/schemas* section and referenced them in the *paths* section using the OAS `$ref` keyword. Since the OAS allows arbitrary schema names as well as inline schemas, for our evaluation, we resolved references in the GT+ OpenAPI descriptions side by side with those in the generated OpenAPI descriptions to match and compare the schemas used for corresponding request bodies, response bodies, and object schema properties. Consequently, schemas used only for request or response bodies that are not detected by an approach cannot be resolved during the comparison and are counted as FN constructs.

We classified a generated *schemas* construct as a TP according to the following criteria, depending on the schema kind. A generated primitive schema is counted as a TP if its type and format match the corresponding GT+ schema. For enum-type schemas, all enum values must match. For array schemas, the item type and the item schema kind must match. For set schemas, additionally, the `uniqueItems` flag must be set. For map schemas, the type and the value's schema kind must match.

Object schemas are counted as TP if all of their object properties are TP and the list of required properties matches. An object property is counted as a TP if it has the correct name and either satisfies the corresponding schema-specific TP criterion or both the generated and GT+ property reference another schema. We did not recursively evaluate such references to avoid cascading FN results from a single incorrect schema, but evaluated the referenced schemas separately. Finally, derived schemas count as TP if a TP object schema extends another object using the OAS `allOf` keyword, both in the GT+ and the generated OpenAPI description.

Additionally, we counted FN constructs, i.e., constructs that only exist in the GT+, and calculated construct coverage as follows:

$$\text{Coverage} = \frac{TP}{TP + FN}.$$

Table 9 shows the coverage results for schema constructs. In the following, we describe the results in detail.

6.2.2. API schemas

AutoOAS detected and reported named schemas in the OpenAPI *#/components/schemas* section and referenced them in the corresponding OpenAPI sections. It correctly identified the formats of simple types, e.g., *integer/int64* for long or *string/date-time* for date-time data types. Furthermore, it accurately reported the OAS schemas describing enums, arrays, sets, maps, and objects. In total, it correctly represented 66% of named schemas used in Spring Boot projects and 52% of named schemas used with JAX-RS. Thereby, it outperformed the second-best approach for Spring Boot, springdoc-openapi, by 30 percentage points, and the second-best approach for JAX-RS, Swagger Core, by 14 percentage points. Next, AutoOAS correctly represented 65% of object schemas for Spring Boot and 49% for JAX-RS, more than double the coverage of the second-best, specialized runtime approaches. Furthermore, AutoOAS was the only approach that retained inheritance information using the OAS `allOf` keyword. It accurately represented 45% of the derived schemas in Spring Boot and 59% in JAX-RS. Notably, missing parameters, request bodies, and response bodies in the generated OpenAPIs automatically entail missing the corresponding schemas.

However, we identified multiple reasons why AutoOAS incorrectly represented schema constructs. First, AutoOAS missed 15 schemas in project Gravitee v4 and one schema in project Quartz because the corresponding classes are located outside the analyzed project directory. Consequently, AutoOAS could not resolve these classes during the source code analysis and only reported an empty object schema with the OAS `externalDocs` information. Additionally, project Quartz contains one data model class whose superclass is located outside the analyzed project directory. Since the derived class itself is located inside the project directory, AutoOAS still parsed the inherited fields and flattened them directly into the object schema, thereby losing inheritance information. The other approaches that analyzed the affected projects, Respector and Swagger Core, did not describe these schemas.

Next, two projects manually serialize response data models with Google's `Gson` library and return the raw JSON string. AutoOAS did not detect this serialization technique and instead reported a string schema, which affected the response schemas in project ProxyPrint and RESTcountries. Similarly, project Senzing manually deserializes JSON string request bodies. Again, AutoOAS did not detect manual deserialization and reported the request body schema to be a string.

Table 9
Overall coverage of schema constructs for the five approaches compared to the GT+ OpenAPI descriptions. Best results are highlighted in **bold**. Overall results average only successful analyses. Unsupported frameworks are marked with a '–'.

Framework		GT+	AutoOAS	Respector	Prophet	springdoc	Swagger Core
Spring Boot	Schemas	111	**0.66**	0.02	0.00	0.36	–
	Obj	93	**0.65**	0.02	0.00	0.32	–
	Derived	11	**0.45**	0.00	0.00	0.00	–
	Props	420	**0.77**	0.06	0.00	0.48	–
JAX-RS	Schemas	197	**0.52**	0.01	–	–	0.38
	Obj	150	**0.49**	0.01	–	–	0.24
	Derived	37	**0.59**	0.00	–	–	0.00
	Props	650	**0.66**	0.10	–	–	0.38

Obj=Object schemas, Derived=Derived schemas, Props=Object schema properties.

Two projects use third-party libraries which are not supported by AutoOAS. Project Digdag uses a Google library for `Optional` objects. While AutoOAS correctly unwraps Java's `Optional`, it incorrectly reported the third-party `Optional` class as the request or response schema containing the `externalDocs` field. Similarly, project EnviroCar uses a third-party library for `DateTime`, which AutoOAS again reported as an empty object with the `externalDocs` field.

As described in Section 5.3.2, project Gravitee v4 uses Jackson's `@JsonValue` field annotation, which replaces the JSON representation of a class with the JSON representation of its annotated field. AutoOAS did not detect these replacement annotations and incorrectly reported the full object schema instead of the field's schema. Contrary, Swagger Core did report the schema of the annotated field.

We found that project OCVN defines multiple inner enum classes with the same name, `Status`. AutoOAS used the class name as a unique identifier and created only a single named enum schema containing the enum values of one enum class. This resulted in multiple missing enum schemas and, additionally, references to the wrong schema.

Finally, AutoOAS reported seven incorrect schemas for request and response body media types other than `application/json`. In particular, the projects ProxyPrint, Cassandra, Digdag, EnviroCar, and Gravitee v4 consume and produce media types such as `image/png` for pictures and `application/yaml` for configuration files. These media types have pre-defined schemas, such as *string/binary* and *string*. AutoOAS incorrectly reported the corresponding schemas as byte arrays or the special *UNSPECIFIED_TYPE* schema. However, we consider this a minor limitation as the schemas of such media types can effectively be ignored.

6.2.3. API schema properties

AutoOAS accurately represented 77% of Spring Boot object schema properties and 66% of properties for JAX-RS, with 29 and 28 percentage points of improvement over the second-best, specialized approaches, springdoc-openapi and Swagger Core, respectively. We reiterate that missing a parameter, request body, or response body in the OpenAPI *paths* section results in missing the corresponding schemas and, hence, all of their properties as well.

Still, AutoOAS missed multiple properties defined by getter methods. The projects EnviroCar and Gravitee v4 use classes with getter methods where only some method names differ from the field names. In such cases, AutoOAS incorrectly reported only the field names. Furthermore, it missed one schema property defined by a computational getter, i.e., the method's return value depends on computations rather than a data field, in project CatWatch, and one abstract getter method in project Features-service. Notably, AutoOAS is the only approach that reported the corresponding object schemas, which are the targets of multiple property schema references.

Answer to RQ3: Representation Accuracy of AutoOAS

AutoOAS's OpenAPI descriptions achieved the highest representation accuracy for operations and responses in Spring Boot and for most constructs in JAX-RS.
Moreover, AutoOAS was the only approach retaining inheritance information of data models and the only static analysis approach considering inherited fields of response objects. It outperformed the second-best approach for Spring Boot, springdoc-openapi, by 30 percentage points in accurate schema representations, and the second-best approach for JAX-RS, Swagger Core, by 14 percentage points.

7. Runtime performance evaluation

In this section, we assess the runtime of AutoOAS and compare it to the state-of-the-art approaches to answer RQ4. We do not consider springdoc-openapi and Swagger Core for this evaluation because they require executing the service under analysis and only expose the OpenAPI description as an additional API operation. Hence, the runtime performance of springdoc-openapi and Swagger Core depends heavily on the service's startup time and the time required to process the API call.

7.1. Evaluation setup

We executed all analyses on a virtual machine running Ubuntu 24.04 LTS, with four available cores of an AMD EPYC 9354 CPU at 3.2 GHz and 16 GB of memory. For each evaluated approach, we analyzed each project five times and report the average runtime per project.

AutoOAS parses the Java source code with Spoon, which is configured to cache project class paths in temporary files. This speeds up repeated analyses of the same project. For this evaluation, we deleted all Spoon cache files before executing each analysis run, i.e., we measured the cold-start time of AutoOAS.

Respector is the only approach that requires bytecode for its static analysis. We compiled all projects before measuring Respector's analysis runtime, i.e., we excluded this preprocessing step from the measurements.

Prophet is published as a Spring Boot web service. It receives the directory path(s) to one or multiple projects as a REST API call parameter and returns the custom JSON format in the response. For this evaluation, we did not consider the one-time effort to start the Prophet web service, and did not measure the time taken for converting the custom JSON format to the OpenAPI description. For each project, we measured only the time required to execute one *curl* command and to write the response body to an output file.

Table 10
Runtimes of the static-analysis approaches for each project in seconds.

Project	Java LoC	AutoOAS	Respector	Prophet
CatWatch	6454	8.8	338.1	0.2
CWA	3616	7.9	4.5	0.1
OCVN	28,099	37.6	6980.0	0.9
Ohsome	10,597	11.1	5814.6	0.5
ProxyPrint	6052	8.9	278.8	0.2
Quartz	3883	7.9	2.4	0.1
Ur-Codebin	1962	7.1	2.1	0.1
Cassandra	8204	9.4	10.3	–
Digdag	90,435	17.6	3.5	–
EnviroCar	25,495	13.0	228.1	–
Features-service	1488	7.0	3.8	–
Gravitee v4	188,671	19.3	7.7	–
Kafka	45,656	14.0	157.6	–
RESTcountries	1605	8.0	13.7	–
Senzing	50,892	13.9	1482.4	–
Overall average	31,540.6	12.8	1021.8	0.3
Overall median	8204.0	9.4	13.7	0.2

7.2. Runtime performance

We present the runtime results in Table 10. Prophet analyzed the projects the fastest, with an average runtime of 0.3 s and a median of 0.2 s. However, Prophet only analyzed handler signatures and did not generate any data model schemas.

AutoOAS obtained an average runtime of 12.8 seconds and a median of 9.4 seconds. It analyzed all projects except for one in less than 20 seconds, and the remaining project, OCVN, in 37.6 seconds. We observed that the runtime of AutoOAS increases approximately linearly with the project size in terms of Java LoC. The only exception is project OCVN, which required more time for analysis than larger projects. We discovered that project OCVN is the only one with an invalid project configuration according to the IntelliJ Integrated Development Environment (IDE). Hence, we expect Spoon to take more time parsing this project correctly. Notably, AutoOAS is the only approach without setup time. It is executable on the command line and only requires the Java source code as input.

Respector was the slowest approach, with an average runtime of 1021.8 s, i.e., 17.0 min, and a median runtime of 13.7 s. However, Respector showed noticeable runtime differences between Spring Boot (average of 32.0 min and median of 4.6 min) and JAX-RS projects (average of 4.0 min and median of 12.0 s). After investigating Respector's evaluation scripts (Huang et al., 2024), we found that the authors optimized the analysis scope to only consider project modules that contain REST API definitions. This applies to the Spring Boot projects CatWatch and Quartz, and the JAX-RS projects Digdag, EnviroCar, Gravitee v4, and Kafka. This manual optimization step for four of the five largest JAX-RS projects explains the overall faster execution times for this framework. We attribute Respector's generally extensive runtime of up to 116.3 min for project OCVN to the symbolic analysis, which identifies parameter constraints that are not part of the OAS.

Answer to RQ4: Runtime of AutoOAS

AutoOAS was slower than Prophet but comparable to or faster than Respector, even with Respector's manually optimized analysis scope.
We consider an average runtime of 12.8 seconds for AutoOAS reasonable, especially given its higher precision, recall, and OpenAPI construct representation accuracy compared to Prophet and Respector.

8. Discussion

In this section, we discuss the results, implications, and the threats to the validity of our evaluations.

8.1. Capabilities and limitations of the approaches

Our evaluations showed different strengths and limitations of the compared approaches. We summarize the capabilities of each approach in Table 11.

Most prominently, AutoOAS and Respector support two Java frameworks, Spring Boot and JAX-RS, while Prophet, springdoc-openapi, and Swagger Core specialize in only one. AutoOAS and Prophet statically analyze the projects' source code, Respector statically analyzes the byte code, and springdoc-openapi and Swagger Core dynamically analyze the services using Java's runtime reflection.

AutoOAS is the only static analysis approach that supports Spring's `@Profile` annotation to detect individual runtime profiles when reporting a REST API's functionality. springdoc-openapi inherently supports Spring profiles because it requires re-running the service under analysis for each profile and only analyzes the corresponding active API operations. However, Spring Boot web services can conditionally expose API operations through mechanisms beyond profiles (Spring, 2026b), which AutoOAS does not resolve. In contrast, springdoc-openapi's runtime reflection correctly captures only the components that are active in the executed runtime configuration. Nonetheless, the analysis would require running the service once for each configuration.

AutoOAS and Swagger Core both consider JAX-RS's `@ApplicationPath` annotation. While AutoOAS reports the declared path prefix in the OAS *#/servers* section, Swagger Core analyzes the exposed operations at runtime, after JAX-RS has already resolved and concatenated the prefix. Similarly, Swagger Core detects class-level parameter annotations because JAX-RS already resolved them at the time of analysis.

Respector symbolically analyzes the byte code to capture arbitrary parameters and responses transmitted as HTTP servlet objects. It is the only approach that detects such constructs, but this comes at the cost of extensive runtimes.

As a major limitation, the dynamic approaches require a valid run configuration and the resources to run the services under analysis. We successfully set up ten projects for dynamic analysis, which took 30.9 min on average (median of 25.5 min). While this setup is a one-time effort, the services must be re-executed for each subsequent analysis, limiting the applicability of the approaches.

Moreover, the dynamic approaches adhere to Swagger's documentation annotations. While this is a design choice, we consider this behavior a limitation. First, giving documentation precedence over the actual implementation holds the risk of outdated OpenAPI descriptions. Second, omitting operations from the OpenAPI description based on annotations that mark them as hidden can make the REST API harder to understand and maintain. We found that some hidden operations expose API management functionality, illustrating that hidden operations do not necessarily correspond to irrelevant functionality.

Regarding the OpenAPI data models, AutoOAS and the dynamic approaches describe object schemas as reusable named schemas in the OpenAPI *#/components/schemas* section. However, springdoc-openapi and Swagger Core flatten all inherited fields into the derived schema, thereby losing inheritance information. In comparison, AutoOAS accurately references the parent schemas using the OAS `allOf` keyword. Respector and Prophet do not use the *#/components/schemas* section. Instead, Respector describes the parameter and response data models inline, and Prophet only reports Java types.

8.2. Implications

AutoOAS offers higher applicability for practitioners than existing approaches for generating OpenAPI descriptions. Prophet, springdoc-openapi, and Swagger Core require additional steps that complicate their usability. In particular, Prophet was published as a web service, where analysis is initiated via a REST API call, and springdoc-openapi

Table 11
Detection capabilities of the five approaches for REST API constructs. The constructs are derived from the shortcomings identified in Section 3 and additional limitations identified during the evaluations of AutoOAS in Sections 5 and 6 (marked with *).

Evaluation dimension	REST API construct	AutoOAS	Respector	Prophet	springdoc	Swagger Core
Framework Support	Spring Boot	✓	✓	✓	✓	✗
	JAX-RS	✓	✓	✗	✗	✓
API Operations	Spring profiles	✓	✗	✗	✓	–
	JAX-RS's ApplicationPath annotation	✓	✗	–	–	✓
	Constants in paths	✓	✓	✗	✓	–
	Regular expression constraints	✓	o	✗	–	–
	Spring's RequestMapping annotation	✓	✓	o	✓	–
API Parameters	Spring's ModelAttribute annotation	✓	✗	✗	–	–
	JAX-RS's FormParam annotation	✓	✓	–	–	✗
	Jackson's JsonProperty annotation	✓	✗	✗	✓	✓
	Request body detection	✓	o	o	✓	o
	Getter methods in abstract classes and interfaces	o	✗	✗	o	o
	HTTP servlet objects*	✗	o	✗	✗	✗
	Advanced Jackson annotations*	✗	✗	–	–	o
	Class-level parameter annotations*	✗	✓	–	–	✓
API Responses	Built-in exception handling	✓	o	✗	✓	✗
	Custom exception handling	o	✗	✗	o	✗
	Default success response codes	✓	o	✗	✓	✗
	Built-in response code methods	✓	✗	✗	–	✗
	Propagated exceptions	o	o	✗	✗	✗
	Propagated response codes*	o	o	✗	o	o
OpenAPI Representation Accuracy	JAX-RS's DefaultValue annotation*	✗	✗	–	–	✓
	Response headers*	✗	✗	✗	o	✗
	Response body media types*	o	✗	✗	o	✗
	OAS data structures	✓	o	✗	o	o
	Object schema inheritance	✓	✗	✗	✗	✗

✓ full support o partial support ✗ no support – not evaluated.

and Swagger Core require executing the service under analysis. Furthermore, Prophet outputs a custom format that cannot be used as an OpenAPI description directly. Respector simplified the generation of OpenAPI descriptions by only requiring the bytecode of the service under analysis. However, it suffers from long runtimes due to its symbolic analysis, limiting its applicability.

In contrast, AutoOAS is implemented as a command line tool that achieves runtimes below 40 seconds while providing high precision, recall, and OpenAPI representation accuracy. In this regard, it outperforms all compared approaches. It particularly improved recall for identifying API parameters and responses compared to existing approaches, which is important for capturing an API's full functionality. Similarly, explicitly referenced and inheritance-aware schemas are necessary for correctly understanding and interacting with a service. First, referencing named schemas in parameters and responses helps developers understand the relationships between calls. Two operations that use the same schemas potentially belong to the same workflow or are API call chain candidates. Second, code generation tools require named schemas to correctly generate API consumers. Without them, they would generate multiple equivalent classes from anonymous schema duplicates, which are incompatible at the type level and therefore not reusable across API call chains.

AutoOAS can be directly integrated into the development process by running it at any time during the implementation phase without requiring bytecode or a valid run configuration. As a practical application, we published AutoGuard (Lercher et al., 2025), a tool for automatically extracting and reporting breaking changes in REST APIs. AutoGuard uses AutoOAS to generate the OpenAPI descriptions of two REST API versions. It relies on accurate descriptions with a high-quality data model representation to identify breaking changes between them. Due to AutoOAS's low runtime, we could fully integrate AutoGuard into GitHub's and GitLab's continuous integration workflows. For every pull and merge request, AutoGuard generates REST API change logs and reports any breaking changes. With this information, developers and code reviewers can make informed decisions on how to proceed with the proposed code changes.

To encourage further applications and use cases, we provide the full source code of AutoOAS in our replication package (Lercher et al., 2026) and publish AutoOAS as a GitHub action and GitLab template,[1] enabling its use in continuous integration workflows.

8.3. Threats to validity

To mitigate threats to **internal validity**, we used an existing dataset with seven Spring Boot and eight JAX-RS web services curated by Huang et al. (2024) for evaluating the precision and recall of API operations, parameters, and response codes detected by AutoOAS. We discovered many mistakes in the dataset's original ground truth document, and the first, second, and third authors inspected the source code of the corresponding projects to discuss and correct the document. To evaluate the OpenAPI representation accuracy, we extended the dataset by manually creating OpenAPI descriptions for a random sample of each project's API operations directly from the source code. The first, second, and fourth authors independently created OpenAPI descriptions for a subset of the projects and discussed all differences until the involved authors reached agreement. Finally, we confirmed that all created OpenAPI descriptions conform to the OAS using two validation tools (OAS Validation, 2026; ApiNotes, 2026). In total, the OpenAPI descriptions comprise 142 operations, 432 parameters, 331 responses, and 308 data model schemas. However, the random sample of API operations may still miss OpenAPI constructs needed to fully describe the projects' REST APIs. We publish our improved ground truth document and manually created OpenAPI descriptions (GT+), the web service projects, the AutoOAS source code, the reported precision, recall, and representation accuracy metrics, and the runtime metrics as a replication package (Lercher et al., 2026) for transparency.

To mitigate threats to **external validity**, we created eight Spring Boot and ten JAX-RS test projects that use typical framework constructs and continuously tested AutoOAS using only these projects during implementation, thereby reducing the risk of bias towards the evaluation

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

dataset. However, while investigating the evaluation dataset as part of our GT+ improvements, we identified four commonly used REST API constructs that were not considered during the initial implementation: Jackson's JsonProperty annotation, JAX-RS sub-resource locators, JAX-RS's ApplicationPath annotation, and JAX-RS's FormParam annotation. We subsequently created five additional test projects covering these constructs and extended AutoOAS accordingly. Consequently, the evaluation dataset influenced the final functionality of AutoOAS. However, we limited the extensions to these four general JAX-RS and serialization constructs. We did not support more specialized constructs such as JAX-RS's class-level parameter annotations (Section 5.3.3), other Jackson annotations (Section 5.3.2), or third-party serialization libraries (Section 6.2.2) to reduce the risk of overfitting AutoOAS to the evaluation dataset. We evaluated AutoOAS on seven Java Spring Boot and eight JAX-RS projects using the improved GT+. The projects feature multiple coding styles, third-party libraries, e.g., Project Lombok (The Project Lombok Authors, 2026), and request handling paradigms, e.g., servlet objects, annotation-based programming, and sub-resource locators. Additionally, the projects comprised varying numbers of REST API paths, HTTP methods, parameters, and responses to be detected. Accordingly, we demonstrated our approach's applicability with the two popular Java web service frameworks Spring Boot and JAX-RS. Still, the AutoOAS implementation does not directly generalize to other Java web frameworks, such as Micronaut (Micronaut Foundation, 2026) and Vert.x (Eclipse Vert.x, 2026), which rely on different annotations or REST API definition mechanisms. Our approach can be extended to other annotation-based Java REST API frameworks by adapting the analyzed annotations and constructs. Finally, we acknowledge that AutoOAS uses Spoon to analyze Java source code. Therefore, it does not support REST APIs implemented in other languages, such as Kotlin.

9. Related work

In this section, we discuss related work on generating OpenAPI descriptions. According to Lercher et al. (2024), OpenAPI is the de-facto standard in the industry to describe REST APIs.

Many existing approaches require executing the service under analysis to generate its OpenAPI description and, hence, require domain knowledge to create a valid run configuration for the service and the infrastructure resources to run it. Yandrapally et al. (2023) proposed ApiCarv, a test suite generation approach that executes an existing UI test suite for a web service in the browser, infers additional UI tests, and generates the OpenAPI description for all identified calls. SwaggerHub Explore (Swagger, 2026b) and AppMap (AppMap, 2026) are commercial products that generate OpenAPI descriptions by recording REST API calls at runtime. Serbout et al. (2023) proposed ExpressO, an approach to generate OpenAPI descriptions for the Express.js JavaScript framework. ExpressO intercepts the initialization stage of a service during startup and stores the loaded operations and their handler source code for analysis.

SpringFox (springfox, 2026) generated OpenAPI descriptions for projects developed with the Java Spring Boot framework. However, the last contribution on GitHub was six years ago, and it was superseded by springdoc-openapi (springdoc, 2026a). Both tools use Java reflection to analyze Spring annotations during service startup and expose the OpenAPI description as an additional service API operation. Both require adding the corresponding dependency to the project. Swagger Core (Swagger, 2026a) generates OpenAPI descriptions for web services developed with the JAX-RS Java framework. Similarly, it uses runtime reflection and exposes an additional API operation for accessing the generated OpenAPI description.

Recently, Huang et al. (2024) introduced Respector, the first approach to generate OpenAPI descriptions by statically analyzing Java bytecode. They compared Respector to AppMap, Swagger Core, SpringFox, and springdoc-openapi, and it outperformed all four alternatives for identifying REST API operations, parameters, and response codes. Additionally, Respector extracts parameter constraints by symbolically analyzing the bytecode and reports them as custom extensions in the OpenAPI description, since the OAS does not support describing such parameter constraints. Cerny et al. (2024) proposed Prophet to visualize microservice architectures reconstructed from the source code. Prophet reports the REST APIs of services in an intermediate format, subsequently used to map API calls to the corresponding service's API operation. For comparison reasons, we converted the intermediate format to OpenAPI descriptions. Deng et al. (2025) were the first to utilize large language models (LLMs) for generating OpenAPI descriptions from the source code. However, their approach LRASGen only reported API response status codes without data model schemas. Moreover, we observed that several LLM-generated OpenAPI descriptions did not adhere to the OAS. For instance, the descriptions specified OAS version 3 but contained multiple API parameters with the *body* location, which is only valid in OAS version 2. We attribute such syntactical errors to potential LLM hallucinations.

To the best of our knowledge, no existing approach can generate OpenAPI descriptions that accurately represent runtime profiles, exception handling, and data models, i.e., object and inheritance information. Furthermore, many existing approaches require injection into the running service to generate the OpenAPI description, which complicates automated analysis. AutoOAS addresses these shortcomings and, as a result, generates more accurate and detailed OpenAPI descriptions.

10. Conclusion

In this paper, we proposed AutoOAS, our approach for automatically generating accurate OpenAPI descriptions of REST API web services from Java source code. AutoOAS addresses current shortcomings of the state-of-the-art approaches Respector, Prophet, springdoc-openapi, and Swagger Core. It is the first static analysis approach that considers Spring's runtime profiles by generating one OpenAPI description per profile configuration, and that considers exception handling by translating potential exceptions in handlers, i.e., Java methods, into HTTP response codes. Moreover, AutoOAS is the first approach that accurately reports parameter and response schemas, i.e., the data models of the REST API. It describes objects as named schemas in the *#/components/schemas* section of the OpenAPI and references them in the corresponding parameter and response descriptions, and accurately models inheritance by reporting separate named schemas for an object's superclasses.

We evaluated AutoOAS on seven Java Spring Boot and eight JAX-RS projects and compared the results to Respector, Prophet, springdoc-openapi, and Swagger Core. The results showed that AutoOAS outperformed the second-best approach, Respector, by 29 percentage points in precision and 27 in recall when identifying API parameters, and by 25 percentage points in precision and 34 in recall when identifying API response codes across all projects. Moreover, AutoOAS was the only approach that retained the inheritance information of data models and the only static analysis approach that considered inherited fields of response objects. It outperformed the second-best approach for Spring Boot, springdoc-openapi, by 30 percentage points in accurate schema representations, and the second-best for JAX-RS, Swagger Core, by 14 percentage points. We observed that AutoOAS achieved these improvements at an average runtime of 12.8 seconds and a maximum runtime of 37.6 seconds. These low runtimes allow AutoOAS to be directly integrated into the development process. Accordingly, we previously published AutoGuard (Lercher et al., 2025) as a practical application of AutoOAS.

Future work concerns handling propagated response codes and exceptions to further improve the precision and recall of our approach. Moreover, we plan to extend AutoOAS to detect inter-parameter dependencies (Martin-Lopez et al., 2019) and to define how to encode them in OpenAPI descriptions.

CRediT authorship contribution statement

Alexander Lercher: Writing – original draft, Visualization, Validation, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **David Jamnig:** Writing – original draft, Validation, Software, Formal analysis. **Christian Macho:** Writing – original draft, Validation, Funding acquisition, Formal analysis. **Clemens Bauer:** Writing – original draft, Software. **Martin Pinzger:** Writing – original draft, Supervision, Funding acquisition.

Generative AI statement

We used generative AI, specifically ChatGPT 5, solely to correct language errors and polish the style of individual sentences.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Given his role as Associate Editor, Martin Pinzger had no involvement in the peer review of this article and had no access to information regarding its peer review. Full responsibility for the editorial process for this article was delegated to another journal editor. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was funded in whole or in part by the Austrian Science Fund (FWF) 10.55776/P36698. For open access purposes, the author has applied a CC BY public copyright license to any author accepted manuscript version arising from this submission.

Data availability

The replication package is available at: https://doi.org/10.5281/zenodo.18877085.

References

APILayer, 2026. REST countries. https://github.com/WebFuzzing/Dataset/tree/master/jdk_8_maven/cs/rest/original/restcountries. (Accessed 12 March 2026).

ApiNotes, 2026. Free online OpenAPI 3.2, 3.1 & swagger validator. https://apinotes.io/openapi-validator. (Accessed 23 June 2026).

APIs You Won't Hate, 2026. OpenAPI.Tools. https://openapi.tools. (Accessed 12 March 2026).

AppMap, 2026. Appmap. https://appmap.io/. (Accessed 12 March 2026).

Arcuri, A., 2019. Restful API automated test case generation with EvoMaster. ACM Trans. Softw. Eng. Methodol. 28 (1), http://dx.doi.org/10.1145/3293455.

Cerny, T., Abdelfattah, A.S., Yero, J., Taibi, D., 2024. From static code analysis to visual models of microservice architecture. Clust. Comput. 27 (4), 4145–4170. http://dx.doi.org/10.1007/s10586-024-04394-7.

Confluent Inc., 2026. Kafka REST proxy. https://github.com/confluentinc/kafka-rest. (Accessed 12 March 2026).

DataStax Inc., 2026. Management API for apache cassandra. https://github.com/k8ssandra/management-api-for-apache-cassandra. (Accessed 12 March 2026).

Deng, S., Huang, R., Zhang, M., Cui, C., Towey, D., Wang, R., 2025. LRASGen: LLM-based restful API specification generation. arXiv:2504.16833. URL: https://arxiv.org/abs/2504.16833.

Deutsche Telekom AG, 2026. Corona-warn-app verification server. https://github.com/WebFuzzing/EMB/tree/master/jdk_11_maven/em/embedded/rest/cwa-verification. (Accessed 12 March 2026).

Development Gateway, 2026. Open contracting Vietnam (OCVN). https://github.com/WebFuzzing/EMB/tree/master/jdk_8_maven/cs/rest-gui/ocvn. (Accessed 12 March 2026).

Eclipse Foundation, 2026a. Annotation type FormParam. https://jakarta.ee/specifications/platform/9/apidocs/jakarta/ws/rs/formparam. (Accessed 12 March 2026).

Eclipse Foundation, 2026b. Jakarta restful web services. https://projects.eclipse.org/projects/ee4j.rest. (Accessed 12 March 2026).

Eclipse Vert.x, 2026. Vert.x-web. https://vertx.io/docs/vertx-web/java/. (Accessed 23 June 2026).

Ed-Douibi, H., Daniel, G., Cabot, J., 2020. OpenAPI bot: A chatbot to help you understand REST APIs. In: Bielikova, M., Mikkonen, T., Pautasso, C. (Eds.), Web Engineering. Springer International Publishing, Cham, pp. 538–542.

Ferrara, J.M., 2026. Features model MicroService. https://github.com/WebFuzzing/Dataset/tree/master/jdk_8_maven/cs/rest/original/features-service. (Accessed 12 March 2026).

Fielding, R.T., 2000. REST: Architectural Styles and the Design of Network-Based Software Architectures (Doctoral dissertation). University of California.

Formosa, F., 2026. Quartz manager. https://github.com/fabioformosa/quartz-manager. (Accessed 12 March 2026).

GIScience Research Group and HeiGIT, 2026. Ohsome API. https://github.com/GIScience/ohsome-api. (Accessed 12 March 2026).

Gravitee.io, 2026. Gravitee.io API management. https://github.com/gravitee-io/gravitee-api-management. (Accessed 12 March 2026).

Huang, R., Motwani, M., Martinez, I., Orso, A., 2024. Generating REST API specifications through static analysis. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3597503.3639137.

Karlsson, S., Čaušević, A., Sundmark, D., 2020. Quickrest: Property-based test generation of openapI-Described restful APIs. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification. ICST, pp. 131–141. http://dx.doi.org/10.1109/ICST46399.2020.00023.

Kim, M., Stennett, T., Shah, D., Sinha, S., Orso, A., 2024. Leveraging large language models to improve REST API testing. In: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results. In: ICSE-NIER'24, Association for Computing Machinery, New York, NY, USA, pp. 37–41. http://dx.doi.org/10.1145/3639476.3639769.

Koren, I., Klamma, R., 2018. The exploitation of openapi documentation for the generation of web frontends. In: Companion Proceedings of the the Web Conference 2018. WWW '18, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, pp. 781–787. http://dx.doi.org/10.1145/3184558.3188740.

Lercher, A., Bauer, C., Macho, C., Pinzger, M., 2025. AutoGuard: Reporting breaking changes of REST APIs from java spring boot source code. In: 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 814–818. http://dx.doi.org/10.1109/SANER64311.2025.00083.

Lercher, A., Glock, J., Macho, C., Pinzger, M., 2024. Microservice API evolution in practice: A study on strategies and challenges. J. Syst. Softw. 215, 112110. http://dx.doi.org/10.1016/j.jss.2024.112110, URL: https://www.sciencedirect.com/science/article/pii/S0164121224001559.

Lercher, A.G., Jamnig, D., Macho, C., Bauer, C., Pinzger, M., 2026. Generating Accurate OpenAPI Descriptions from Java Source Code - Replication Package. Zenodo, http://dx.doi.org/10.5281/zenodo.18877085.

Martin-Lopez, A., Segura, S., Ruiz-Cortés, A., 2019. A catalogue of inter-parameter dependencies in restful web APIs. In: Yangui, S., Bouassida Rodriguez, I., Drira, K., Tari, Z. (Eds.), Service-Oriented Computing. Springer International Publishing, Cham, pp. 399–414.

Mathew Estafanous, 2026. Ur-codebin. https://github.com/Mathew-Estafanous/Ur-Codebin-API. (Accessed 12 March 2026).

Micronaut Foundation, 2026. Micronaut. https://micronaut.io/. (Accessed 23 June 2026).

OAS Validation, 2026. Validate your openapi specs. https://oas-validation.com/. (Accessed 23 June 2026).

OpenAPI Initiative, 2026a. Components object. https://spec.openapis.org/oas/v3.1.0.html#components-object. (Accessed 12 March 2026).

OpenAPI Initiative, 2026b. Data types. https://spec.openapis.org/oas/v3.1.0.html#data-types. (Accessed 12 March 2026).

OpenAPI Initiative, 2026c. Model with map/dictionary properties. https://spec.openapis.org/oas/v3.1.0.html#model-with-map-dictionary-properties. (Accessed 12 March 2026).

OpenAPI Initiative, 2026d. OpenAPI specification v3.1.0. https://spec.openapis.org/oas/v3.1.0.html. (Accessed 12 March 2026).

OpenAPI Initiative, 2026e. Responses object. https://spec.openapis.org/oas/v3.1.0.html#responses-object. (Accessed 12 March 2026).

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L., 2015. Spoon: A library for implementing analyses and transformations of java source code. Softw.: Pr. Exp. 46, 1155–1179. http://dx.doi.org/10.1002/spe.2346, URL: https://hal.archives-ouvertes.fr/hal-01078532/document.

Peng, C., Goswami, P., Bai, G., 2018. Fuzzy matching of openapi described REST services. Procedia Comput. Sci. 126, 1313–1322. http://dx.doi.org/10.1016/j.procs.2018.08.081, URL: https://www.sciencedirect.com/science/article/pii/S1877050918313590. Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 22nd International Conference, KES-2018, Belgrade, Serbia.

ProxyPrint, 2026. Proxyprint-kitchen. https://github.com/WebFuzzing/EMB/tree/master/jdk_8_maven/cs/rest/original/proxyprint. (Accessed 12 March 2026).
Senzing community, 2026. Senzing REST API. https://github.com/senzing-garage/senzing-api-server. (Accessed 12 March 2026).
Serbout, S., Romanelli, A., Pautasso, C., 2023. Expresso: From express.js implementation code to OpenAPI interface descriptions. In: Batista, T., Bureš, T., Raibulet, C., Muccini, H. (Eds.), Software Architecture. ECSA 2022 Tracks and Workshops. Springer International Publishing, Cham, pp. 29–44.
Spring, 2026a. Annotated controllers. https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller.html. (Accessed 12 March 2026).
Spring, 2026b. Annotation interface conditional. https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Conditional.html. (Accessed 23 June 2026).
Spring, 2026c. @Modelattribute. https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/modelattrib-method-args.html. (Accessed 12 March 2026).
Spring, 2026d. Profiles. https://docs.spring.io/spring-boot/reference/features/profiles.html. (Accessed 12 March 2026).
Spring, 2026e. Spring boot. https://spring.io/projects/spring-boot. (Accessed 12 March 2026).
springdoc, 2026a. Springdoc-openapi. https://github.com/springdoc/springdoc-openapi. (Accessed 12 March 2026).
springdoc, 2026b. What are the ignored types in the documentation? https://springdoc.org/#what-are-the-ignored-types-in-the-documentation. (Accessed 12 March 2026).
springfox, 2026. Springfox. https://github.com/springfox/springfox. (Accessed 12 March 2026).
Swagger, 2026a. Swagger core. https://github.com/swagger-api/swagger-core. (Accessed 12 March 2026).
Swagger, 2026b. Swaggerhub explore. https://swagger.io/tools/swaggerhub-explore/. (Accessed 12 March 2026).
The enviroCar Project, 2026. Envirocar server. https://github.com/enviroCar/enviroCar-server. (Accessed 12 March 2026).
The Project Lombok Authors, 2026. Project lombok. https://projectlombok.org/. (Accessed 12 March 2026).
The Zalando Incubator, 2026. CatWatch. https://github.com/WebFuzzing/EMB/tree/master/jdk_8_maven/cs/rest/original/catwatch. (Accessed 12 March 2026).
Treasure Data, 2026. Digdag. https://github.com/treasure-data/digdag. (Accessed 12 March 2026).
Yandrapally, R., Sinha, S., Tzoref-Brill, R., Mesbah, A., 2023. Carving UI tests to generate API tests and API specification. In: Proceedings of the 45th International Conference on Software Engineering. ICSE '23, IEEE Press, pp. 1971–1982. http://dx.doi.org/10.1109/ICSE48619.2023.00167.

Alexander Lercher is a Ph.D. student in the Software Engineering Research Group at the University of Klagenfurt, Austria. He received his Master's degree (Dipl.-Ing.) with a focus on distributed systems from the University of Klagenfurt in 2021 with distinction. His research interests include microservice architectures, software and API evolution, and program analysis.

David Jamnig is a research assistant in the Software Engineering Research Group at the University of Klagenfurt, where he completed his Master's degree (Dipl.-Ing.) in Informatics in 2026 with distinction. His research focuses on web API evolution, program analysis, change impact analysis, and empirical software engineering.

Christian Macho is a PostDoc assistant (tenure track) in the Software Engineering Research Group (SERG) at the University of Klagenfurt. He received his M.Sc. from the Technical University Vienna in March 2015 and his Ph.D. from the University of Klagenfurt in 2019 both with distinction. His research interests include software evolution, mining software repositories, program analysis, build systems, continuous integration, automated repair, and empirical studies in software engineering.

Clemens Bauer is a research assistant in the Software Engineering Research Group at the University of Klagenfurt, Austria. He received his Master's degree (Dipl.-Ing.) in Informatics, with a focus on software engineering, from the University of Klagenfurt in 2026, with distinction. His research interests include program analysis, program testing, and test automation.

Martin Pinzger is a full professor at the University of Klagenfurt, Austria where he is heading the Software Engineering Research Group. His research interests are in software evolution, mining software repositories, program analysis, software visualization, and automating software engineering tasks. He is a member of ACM and a senior member of IEEE.