Preprocessing of the 20th Conference on Computer
Science and Intelligence Systems (FedCSIS) pp. 343–348

Enhancing API documentation by inter-endpoint dependency graphs

Panagiotis Papadeas
0009-0003-3012-0089
Software Engineering Lab, National Technical University of Athens. Zografos 15780, Greece
Email: papadeas@cs.ntua.gr

Dimitrios Gerokonstantis
0009-0004-0950-8414
Software Engineering Lab, National Technical University of Athens. Zografos 15780, Greece
Email: dgerokonstantis@cs.ntua.gr

Christos Hadjichristofi
0009-0000-7976-8749
Software Engineering Lab, National Technical University of Athens. Zografos 15780, Greece
Email: hadjichristofi_ch@mail.ntua.gr

Vassilios Vescoukis
0000-0002-5360-8349
Software Engineering Lab and Geoinformation Center, National Technical University of Athens. Zografos 15780, Greece
Email: v.vescoukis@cs.ntua.gr

***Abstract*—APIs are the main mechanism for integrating heterogeneous components in modern software systems. API documentation typically treats each endpoint as autonomous, leaving out information about other endpoints that produce or consume input or output data, which is useful in rapidly orchestrating valid call sequences. This makes the development, testing and maintenance of systems that use APIs to internally implement business logic, or integrate with third-party API-based services, more complex. This paper addresses this gap by introducing static and dynamic analysis methods to automatically discover and document inter-endpoint dependencies; static analysis uses existing API documentation, while dynamic analysis is based on logging API usage. Both methods are integrated into RADAR, a tool for extracting and visualizing graphs, to illustrate such inter-endpoint dependencies through exchanged data. Case study on the PayPal API demonstrates the value of adding dependency graphs to API documentation to support rapid development and documentation of software systems.**

***Index Terms*—APIs, inter-endpoint dependencies, API documentation.**

I. Introduction

IN MODERN software architectures, Application Programming Interfaces (APIs) serve as the primary mechanism for exposing the functionalities offered by entire software applications or even components, to other applications, components, or services of any kind. An API exposes a set of endpoints that, when invoked by a consumer, execute specific server-side functionality, which results in a response sent back to the client, regardless of the technologies used on either side.

APIs are developed both for integrating components within applications and as standalone software products. In either case, their complexity necessitates comprehensive documentation to support their development, maintenance, and integration. Typical API documentation usually follows standards such as OpenAPI[1], or comes in the form of de-facto formats introduced by tools, such as Postman[2] collections. Any such kind of documentation provides the schema of the request and response of every endpoint, data types and descriptions, and possibly full examples of API calls with data.

However, it usually treats each endpoint as an independent service, whereas an API is a collection of interacting endpoints that collectively can be used to implement pieces of business logic. To do this effectively, it is essential that developers have information not only about how to call each endpoint, but also about all its possible interactions with other endpoints that either consume output or provide input data. This work focuses on the discovery, modeling and representation of such interactions in a way that provides additional practically useful documentation for the integration of APIs into software components of any kind.

II. Related Work

Prior research has explored ways to facilitate the proper utilization of APIs by identifying the constraints that must be met when interacting with them. A wide area of research investigates techniques for identifying interdependencies among input parameters within (single) API calls, with the goal of facilitating the construction of syntactically and semantically valid API requests.

Oostvogels et al. [1] attempted to analyze and incorporate inter-parameter constraints within API documentation. These constraints define rules whereby the existence or valid values of certain input parameters in an API call depend on the presence or values of other parameters. To address the absence of structured representation for such constraints in existing documentation, a new specification language was created as an extension to the OpenAPI specification to incorporate this information. Martin-Lopez et al. ([2], [3], [4]) also focused on inter-parameter dependencies, proposing a Domain Specific

[1] OpenAPI: https://www.openapis.org/

[2] Postman: https://www.postman.com/

Language known as IDL (Inter-parameter Dependency Language) to describe these dependencies. Both studies focus on interdependencies of input parameters for a specific call rather than on inter-endpoint dependencies which is our area of interest.

Wu et al. [5] proposed an additional method that detects "dependency constraints on parameters", leading to the development of the INDICATOR system, which analyzes documentation and SDKs to identify constraint candidates and then validates them through testing. The INDICATOR system investigates these producer-consumer relationships based on the names and types of parameters. However, it does not focus on systematically extracting or documenting these relationships.

Another related area of research concentrates on identifying interdependencies across different operations of Web Services and APIs for testing. The work of Bai et al. [6] aims to automatically generate test cases from WSDL specifications with respect to the dependencies among various operations. One of the dependency types examined is input/output dependencies (IOD), which represent inter-endpoint dependencies. However, this approach limits the examination of inter-endpoint dependencies to aspects directly related to test case generation, which aligns with the focus of their study.

The study of Atlidakis et al. [7] introduces REST-ler, a tool that analyzes Swagger specifications for RESTful APIs, automatically identifies object types required by a request that have been produced by previous ones, generates tests with respect to these dependencies and dynamically infers which request sequences are valid. However, this inter-endpoint analysis aims mainly to automatically generate and execute meaningful and correct test cases, while adhering to these dependencies. Consequently, no documentation regarding the inter-endpoint dependencies is generated.

The study conducted by Bertolino et al. [8] aims to extract behavioral information regarding how users should invoke a Web Service. This method, called StrawBerry, examines the WSDL document and identifies dependencies between the operations of the Web Service based solely on types of input and output parameters. A behavior protocol automaton is generated, indicating the dependencies and data transfers among various operations of the Web Service. However, this analysis relies on the presence of a WSDL specification.

III. Inter-Endpoint dependencies

An "inter-endpoint dependency" is an interaction between API endpoints. If invoking endpoint Y requires input data received from a call to endpoint X, then the required call order is (X, Y) and thus we say that Y "depends" on X. As an example, consider the hypothetical Flight Booking API illustrated in Fig 1, along with a sample invocation of its endpoints. The top-level nodes represent API endpoints, while the corresponding JSON response schemas and the example responses appear beneath each endpoint. Directed edges indicate the data flow between endpoints.

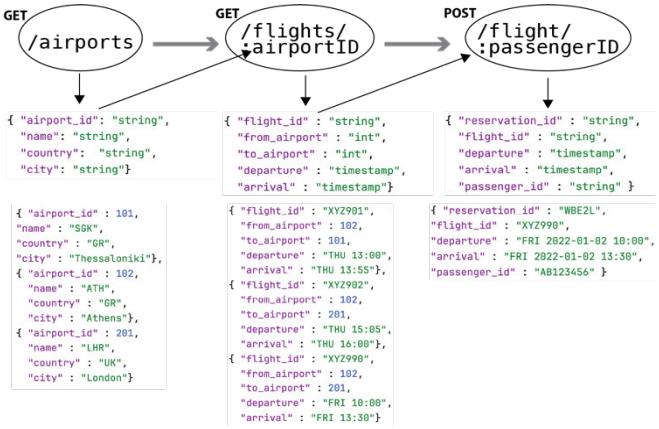

Fig 1. Production and consumption of data in API call sequences

The call to the /airports endpoint returns a list with all available airports, including their identifiers (airport_id). The following call to /flights takes as input, as a path parameter, the id of one of these airports (102, as shown in the example) and returns information about available flights departing from that airport. This is a body-path dependency, as a data element from the response body of the first call is used as a path parameter in the second call.

Subsequently, the /flight endpoint call takes as input, as the request body this time, all the details of one of the flights returned by the second call (the one with id=XYZ990, in the example) along with a path parameter that points to a passenger and performs the booking for that flight and passenger. This represents a body-body dependency, as the request body of the third call requires a data object included in the response body of the second call. Similarly, in body-query dependencies the value of a query parameter comes from a value contained in some attribute of a response body from a previous call. Notably, inter-endpoint dependencies can be identified by more than one attributes.

Examining inter-endpoint dependencies goes beyond typical API documentation by revealing the correct sequences and prerequisites of API calls. This knowledge eases the implementation of requests with valid inputs and helps discover entire paths that indicate the semantically correct order of endpoint invocation. As such, it can accelerate integration with third-party APIs by identifying valid call sequences. In microservice architectures, it also supports—and can even partially automate—the orchestration control flow. Finally, for API developers, it serves as a valuable reference for understanding data flow, maintaining consistency, and onboarding new team members.

This work focuses on the discovery and documentation of API inter-endpoint dependencies by introducing algorithms as well as a tool to discover, analyze, and visualize them using a directed graph, whose nodes represent the API endpoints and each directed edge represents the complete or partial consumption of data returned by the origin, by the destination endpoint.

IV. Discovery of Dependencies

Two approaches are demonstrated for discovering and analyzing inter-endpoint dependencies: static and dynamic, introduced in the diploma theses of Papadeas [9] and Gerokonstantis [10], respectively. The static approach is based on existing API documentation, whereas the dynamic captures and analyzes API calls made during the real-world utilization of the API.

4.1 Static analysis

Static analysis is entirely based on the information included in the API documentation. It aims to identify inter-endpoint dependencies using this documentation (structure, types), which typically contains details about the input and output parameters of each endpoint. The goal is to identify dependencies and generate a directed graph that shows all possible endpoint call sequences.

Static analysis takes its input from any standard API documentation, such as OpenAPI, or even from ad-hoc documentation such as Postman, Apidog[3], Firecamp[4] or other similar tools. It infers the proper sequence of API calls by analyzing the results of pre-executed calls included in the documentation examples. Lack of pre-executed calls and insufficient examples reduces the capability of this analysis to discover dependencies.

In our implementation, static analysis takes its input from a Postman collection to create a representation of the directed dependency graph. The Postman collection comes in as a JSON file that contains information about responses, request body, path and query parameters of each endpoint. The use of Postman collections has been chosen for illustration purposes only: any structured parsable object containing the same data can be used instead, and the results will still rely on the completeness of metadata included in the documentation.

A typical context where static analysis can be performed, is when we have some kind of documentation about an API, and we don't know anything about the proper sequence of endpoint calls or possible use cases of the API. The graph produced by the static analysis helps us get a first view of the API usage flow, as it shows how endpoints relate to each other.

The algorithm of the static analysis takes as input a postman collection file (Algorithm 1). For every endpoint of the API it stores information about its parameters including name, value and type of each parameter in a set named ParametersSet (lines 1-5 Algorithm 1). It also stores the response of each endpoint in a separate set named AttributesSet including the name, value and type of each attribute in the response (lines 6-10 Algorithm 1). Then it proceeds to compare the two sets to discover dependencies (lines 11-12 Algorithm 1).

Each dependency can be identified either by matching the values of parameters with the values of responses exchanged between endpoints in call examples (Algorithm 2), or by matching keys instead (Algorithm 3).

The algorithm compares the two sets to find every pair of attributes and parameters that have the same value (or name, respectively), type and are not part of the same endpoint (lines 1-4 Algorithm 2 and Algorithm 3 respectively). For each case where a match has been identified, a new edge is added to the graph, along with the relevant information regarding the dependent endpoints (lines 5-8 Algorithm 2, 3).

Algorithm 1. Static inter-endpoint dependency analysis

```
 1. Read postman_collection.json as Input
 2.
 3. foreach endpoint in Input do
 4.     foreach parameter in endpoint do
 5.         ADD(parameter(name, value, type)) TO SET
            named ParametersSet
 8.
 9. foreach endpoint in Input do
10.     foreach response in endpoint do
11.         ADD(attribute(name, value, type)) TO SET
            named AttributesSet
14.
15. DependencyGraphByValues(ParametersSet, AttributesSet)
16. DependencyGraphByNames(ParametersSet, AttributesSet)
```

Algorithm 2. DependencyGraphByValues

```
1. Procedure DependencyGraphByValues
2.
3. foreach parameter in ParametersSet do
4.     foreach attribute in AttributesSet do
5.         if parameter.value == attribute.value AND
              parameter.type == attribute.type AND
              endpoint(parameter) != endpoint(attribute)
           then
6.            ADD(dependency(endpoint(parameter),
                      endpoint(attribute)))
              TO LIST named Dependencies
7.
8. return Dependencies
```

Algorithm 3. DependencyGraphByNames

```
 1. Procedure DependencyGraphByNames
 2.
 3. foreach parameter in ParametersSet do
 4.     foreach attribute in AttributesSet do
 5.         if parameter.name == attribute.name AND
 6.            parameter.type == attribute.type AND
 7.            endpoint(parameter) != endpoint(attribute)
           then
 8.                ADD(dependency(endpoint(parameter),
                           endpoint(attribute)))
                   TO LIST named Dependencies
12.
13. return Dependencies
```

The second way of dependency identification, by comparing only the keys of parameters, has been proven to be useful in cases where the collection does not contain many examples of pre-executed calls and there isn't enough information regarding each endpoint. In that case, the only way a dependency could be inferred is by comparing keys between parameters and responses assuming that common keys could also imply dependent values.

Static analysis will usually discover a greater-than-actual number of inter-endpoint dependencies due to common key-value patterns across endpoints. For example, the frequent use of generic keys like "ID" can falsely suggest dependencies between unrelated endpoints. These findings are often hard to evaluate without access to actual API calls or a good understanding of the API's semantics. The goal of this analysis is to

[3] Apidog: API Development Tool - https://apidog.com

[4] Firecamp: Open Source Postman Alternative - https://firecamp.io

identify all possible dependencies, including dependencies which are considered “noise”.

Dynamic analysis

The dynamic analysis takes a different approach, aiming to discover inter-endpoint dependencies without using any kind of documentation. It collects information from HTTP requests and responses, by monitoring actual calls using a "man-in-the-middle" agent that captures HTTP traffic. The API traffic is logged and then the extraction of inter-endpoint dependencies is possible. Notably, this method follows a reverse engineering approach, analyzing already implemented but still undocumented APIs, to infer inter-endpoint dependencies.

Dynamic analysis also identifies all types of dependencies, just as static analysis does. However, in the case of body-path dependencies, since it is unclear which segments of a URL correspond to path parameters, the algorithm treats all URL segments as such. This allows it to identify the corresponding dependencies, introducing minimal noise.

In cases where the API is consumed through a web frontend, API calls can be captured by executing use cases while interacting with the frontend in a browser. In the context of an API ecosystem implementing orchestration, as in a microservices architecture, it is possible to capture API calls as part of the interaction between the orchestrator component and the internal components of the architecture.

In any of the approaches mentioned above, to capture the HTTP traffic resulting from real-world interactions with the API, a "mediator" is required, that can be implemented in several ways, such as the following:

1. a standalone mediator component to which API calls would be directed, responsible for forwarding the requests to the API server, returning the responses to the client, and recording information about the API calls; such a system was implemented in the context of this paper,

2. a browser extension that captures the incoming and outgoing traffic between the browser and the API while the user navigates through the User Interface interacting with the API,

3. an internal component within API Gateways, enabling the collection of multi-client data derived from the interactions of multiple external entities with the API.

Similar to the static method, the outcome of the dynamic analysis is a directed dependency graph of the same type. The input is a log file generated by the mediator, which contains information about the executed API calls.

The dynamic analysis algorithm comprises two main components: the parsing of the log file and extraction of value-specific information (Algorithm 4), and the identification of matches among the extracted values (Algorithm 5). Specifically, Algorithm 4 processes the input log file, assigns a sequence number to each API Call reflecting its execution order within the use case it belongs to, and constructs two dictionaries, REQUEST_VALUES and RESPONSE_VALUES, which store information about every value encountered in API request and response messages respectively. For each identified value, the dictionaries record the associated API call’s URL and sequence number, the identifier ("tag") of the corresponding use case, the value’s name, and its inferred data type.

The dictionaries constructed in Algorithm 4 serve as inputs to Algorithm 5, which is responsible for identifying matches between values appearing in request and response messages. Specifically, for each value V found in the response body of an API call RES, the algorithm examines whether V also appears as a request parameter (of any type) in any subsequent API calls (REQ_1, REQ_2, ..., REQ_n) within the same use case.

If such matches are found, the algorithm adds a directed dependency edge from RES to each matching REQ_i, provided the following conditions are met:

- the data type of V matches in both RES and REQ_i, as determined by type inference,
- RES and REQ_i correspond to distinct API endpoints, and
- REQ_i occurs after RES in the execution order of the use case.

Algorithm 4. Dynamic inter-endpoint dependency analysis

```
1. Initialization: Assign sequence numbers to the
   calls of every use case
2.
3. foreach call in Input do
4.     if call.body then
5.         foreach parameter in call.body do
6.             REQ_VALUES[parameter.value].APPEND({
7.               url: call.url, seq: call.sequence_number,
8.               tag: call.tag, paramName: parameter.name,
9.               paramType: parameter.type})
10.
11.    if call.response then
12.        foreach attribute in call.response do
13.            RES_VALUES[attribute.value].APPEND({
14.              url: call.url, seq: call.sequence_number,
15.              tag: call.tag, attrName: attribute.name,
16.              attrType: attribute.type})
17.
18. ComputeDependencyGraph(REQ_VALUES, RES_VALUES)
```

Algorithm 5. ComputeDependencyGraph

```
1.   Procedure ComputeDependencyGraph
2.     foreach value in RES_VALUES do
3.       if value in REQ_VALUES then
4.         foreach appearRes in RES_VALUES[value] do
5.           foreach appearReq in REQ_VALUES[value] do
6.             if appearRes.paramType ==
                  appearReq.paramType and
7.                appearRes.url ≠ appearReq.url and
8.                appearRes.seq < appearReq.seq and
9.                appearRes.tag == appearReq.tag
10.            then
11.            DEPENDENCY_GRAPH[
                      (appearRes.url, appearReq.url)
               ].APPEND({
12.              fromAttribute: appearRes.paramName,
13.              toAttribute: appearReq.paramName
14.            })
```

A key difference from the static method is that dynamic analysis restricts comparisons to subsequent API calls of the same use case, focusing on the dependencies most likely to be semantically correct within the context of the specific API.

While dynamic analysis can generate dependency documentation even for completely undocumented APIs, it also has several challenges, with the primary one being that the discovery of all dependencies requires the execution of calls between all dependent endpoints, which may not occur during the logging.

For visualizing the resulting dependency graph, a tool named RADAR[5] (REST API Dependencies and Analysis of Relationships) was implemented [11] and is available at radar.softlab.ntua.gr. RADAR takes as input either the Postman Collection of the API or a log file containing records of actual API calls, for static and dynamic analysis, respectively. After the user configures certain analysis parameters, RADAR performs either static or dynamic analysis to detect inter-endpoint dependencies, and generates a directed dependency graph, as discussed above.

V. Case Study

The two dependency analysis methods were applied, among others, to the PayPal API sandbox environment[6].

The PayPal API provides a wide range of functionalities, including the creation and management of products, orders, invoices, subscriptions, payments, and disputes. The complexity and size of such an API necessitate conducting dependency analysis to facilitate and guide its utilization. Below, we examine how static and dynamic inter-endpoint dependency analysis can assist a user of this API in creating and tracking a product order.

Static

For static analysis, we supply RADAR with the Postman Collection of the PayPal API[7], as downloaded from the Postman Public API Network, without any modifications. The section of the resulting dependency graph including endpoints related to product orders is shown in Fig 2 below. While the identified dependency path facilitates the process of creating an order and handling the payment, it lacks details regarding order tracking. Static analysis was unable to identify dependencies between the tracking endpoints and the others in the graph, as the examples in the Postman Collection were not adequate. Additionally, the static analysis graph contains some edges that are either redundant for executing the specific use case or even semantically incorrect.

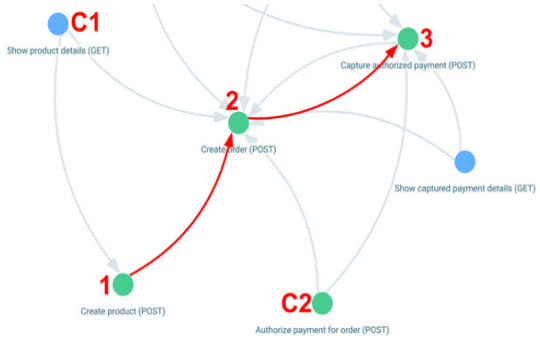

Fig 2. Order creation and tracking use case - Static dependency graph

For instance, displaying product details before creating a new product (Fig 2, edge C1→1) offers no practical value in the context of the "order management" use case, while authorizing a payment for an order before the order itself has been created, as the graph suggests (Fig 2, edge C2→2), seems not correct and should rather be classified as noise. As demonstrated below, dynamic analysis mitigates these issues.

Dynamic

Dynamic analysis involves logging interactions with the API relevant to product orders. To do this, we set up an example use case, in which we begin by creating/updating a product and generating an order for this. After that, we initiate the payment process by authorizing and capturing the payment, and conclude by adding tracking information for the order. This sequence is illustrated in Fig 3.

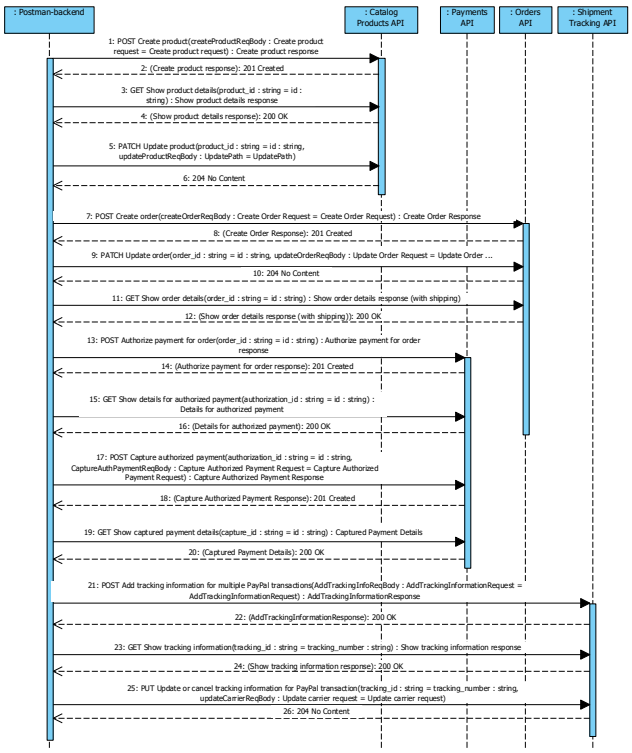

Fig 3. Order creation and tracking use case - sequence diagram

The dependency graph resulting from the dynamic analysis of the log file for this use case is shown in Fig 4, where API calls are enumerated based on their numbering in the sequence diagram above, and a key dependency path is highlighted.

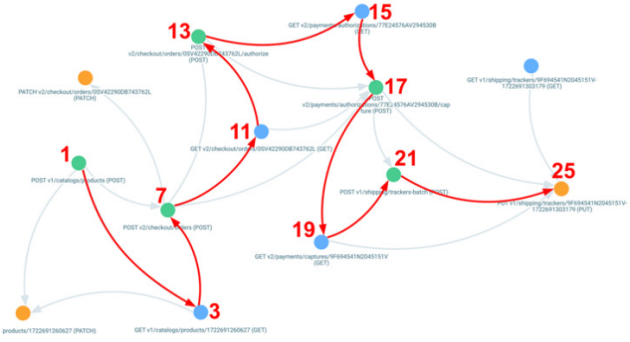

Fig 4. Order creation and tracking use case - dynamic dependency graph

[5] R.A.D.A.R by NTUA Softlab: https://radar.softlab.ntua.gr

[6] PayPal REST APIs: https://developer.paypal.com/api/rest (date accessed: 05/2024)

[7] PayPal API Postman Collection (date accessed: 05/2024): https://www.postman.com/paypal/paypal-public-api-workspace/overview

The highlighted dependency path can be traced back to the reference use case sequence diagram, providing the API user with guidance on how to use the API to implement the logic of the "order management" use case.

Specifically, the generated graph guides the user to first create a product and optionally retrieve information about it (calls 1, 3), then create an order for this product and possibly retrieve relative details (calls 7, 11), authorize and capture the payment (calls 13–19), and finally add and potentially update tracking information (calls 21, 23). In contrast to static analysis, the dynamic dependency graph proved to be more comprehensive while avoiding misleading edges.

VI. Discussion

Like all software documentation, API documentation must be complete, detailed, and kept up to date to support communication among technical staff and stakeholders. As APIs play a central role in software integration, better documentation leads to more efficient development and fewer errors and backtracks.

This work aims to enhance API documentation by adding information on data producers and consumers among endpoints, representing their dependencies. Such an "endpoint dependency graph" helps developers better understand and integrate complex third-party or in-house APIs. We propose two methods to identify these dependencies based on available documentation and runtime access, providing a roadmap to accelerate API integration.

The effectiveness of static analysis in identifying dependencies relies on the information in the API's static documentation, particularly the examples included. On the other hand, the effectiveness of dynamic analysis in the same task depends on the extent to which the executed use cases cover a broad range of the API's functionalities, theoretically all of them.

Generally, the dynamic approach produces less, yet not zero, noise, particularly in body-path dependencies derived from URL segments that are not actually path parameters. In static analysis, which runs exhaustive comparisons between every pair of endpoints, noise is created by ambiguities that do not allow to filter out irrelevant dependencies.

Overall, by examining the statistics derived from the results of the two analyses across the two APIs, we observe that dynamic analysis detects fewer edges in the generated graph: 52 vs. 92 for the OpenAI API and 209 vs. 724 for the PayPal API. Furthermore, dynamic analysis identifies more dependent attributes for each edge, offering a deeper understanding of how two endpoints are interdependent. For example, for the OpenAI API, an average of 1.22 dependent attributes per edge was found in the static analysis and 2.53 in the dynamic analysis, while for the PayPal API, 2.13 dependent attributes per edge were found in the static analysis and 14.19 in the dynamic analysis.

Considering the limitations of each method, as well as the conclusions drawn from their comparative study and application to the API examined, it becomes clear when each method is most suitable: when any static documentation such as a Postman Collection with enough examples is available, dependency documentation for the API can be generated using static analysis. When logging is possible, dynamic analysis can offer higher accuracy and completeness of the results in the context of the observed use cases, but at the cost of greater effort that comes with logging. RADAR supports both methods and although it has space for improvement, it produces useful data and visualizations of inter-endpoint API dependencies.

References

[1] N. Oostvogels, J. De Koster, and W. De Meuter, "Inter-parameter Constraints in Contemporary Web APIs," 2017, pp. 323–335. doi: 10.1007/978-3-319-60131-1_18.

[2] A. Martin-Lopez, "Automated analysis of inter-parameter dependencies in web APIs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, New York, NY, USA: ACM, Jun. 2020, pp. 140–142. doi: 10.1145/3377812.3382173.

[3] A. Martin-Lopez, S. Segura, C. Muller, and A. Ruiz-Cortes, "Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs," *IEEE Trans Serv Comput*, vol. 15, no. 4, pp. 2342–2355, Jul. 2022, doi: 10.1109/TSC.2021.3050610.

[4] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "A Catalogue of Inter-parameter Dependencies in RESTful Web APIs," in *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings*, Berlin, Heidelberg: Springer-Verlag, 2019, pp. 399–414. doi: 10.1007/978-3-030-33702-5_31.

[5] Q. Wu, L. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei, "Inferring dependency constraints on parameters for web services," in *Proceedings of the 22nd international conference on World Wide Web*, in WWW '13. New York, NY, USA: ACM, May 2013, pp. 1421–1432. doi: 10.1145/2488388.2488512.

[6] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen, "WSDL-Based Automatic Test Case Generation for Web Services Testing," in *IEEE International Workshop on Service-Oriented System Engineering (SOSE'05)*, IEEE, 2005, pp. 215–220. doi: 10.1109/SOSE.2005.43.

[7] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API Fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, May 2019, pp. 748–758. doi: 10.1109/ICSE.2019.00083.

[8] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli, "Automatic synthesis of behavior protocols for composable web-services," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, in ESEC/FSE '09. New York, NY, USA: ACM, Aug. 2009, pp. 141–150. doi: 10.1145/1595696.1595719.

[9] P. Papadeas, "Automatic generation of dependency documentation between endpoint calls of a REST API," National Technical University of Athens, 2023. Accessed: Apr. 27, 2025. [Online]. Available: http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/18728

[10] D.-D. Gerokonstantis, "Dynamic Analysis of Inter-endpoint Dependencies in RESTful APIs," National Technical University of Athens, 2024. Accessed: Apr. 27, 2025. [Online]. Available: http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/19419

[11] D. Lalias, "Web application for visualizing dependencies between endpoints of a REST interface," National Technical University of Athens, 2024. Accessed: Apr. 27, 2025. [Online]. Available: http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/19020