Detecting Usage of Deprecated Web APIs via Tracing

Leif Bonorden
Universität Hamburg
Hamburg, Germany
leif.bonorden@uni-hamburg.de
ORCID: 0000-0002-2131-7790

André van Hoorn
Universität Hamburg
Hamburg, Germany
andre.van.hoorn@uni-hamburg.de
ORCID: 0000-0003-2567-6077

***Abstract*—Deprecation is a way to inform clients using an application programming interface (API) that the usage of this API is discouraged. Tool support and research for deprecation in local APIs are well established. However, nowadays web APIs are more commonly used, e.g., using the REST architectural style. However, the techniques to detect and handle the usage of deprecated local APIs cannot be directly applied to web APIs. Previous approaches for detecting deprecated web APIs focus on static analysis of client code by detecting calls to web APIs and, subsequently, an investigation of associated API specifications. These approaches currently have two essential limitations: (i) The target of an API call can often not be determined statically. (ii) Deprecation in API specifications is not the only way to signal deprecation for web APIs.**

We introduce a dynamic approach using tracing to detect calls to web APIs. Subsequently, we check the called APIs for deprecation using an API specification, response meta-data, or a knowledge base. This approach addresses both limitations of the detection with static analysis. We implement the approach and evaluate it on three projects, including client-server calls as well as a microservice benchmark system. The empirical evaluation yields a precision of 1.00 and a recall of 0.95. The false negatives can be attributed to a shortcoming in the automatic instrumentation provided by OpenTelemetry observability framework.

***Index Terms*—application programming interface, deprecation, dynamic analysis, tracing**

I. Introduction

Software systems are typically not isolated units but interact with their surroundings [1]. Such communication with external systems and their interfaces may be a business requirement, thus posing an *architecturally significant requirement*, or a voluntary decision during the system's design, thus introducing the dependency as an *architectural constraint* for further development [2], [3]. In addition to systems external not belonging to the same organization, a similar setting is encountered within a system if it comprises highly decoupled modules, e.g., with a microservice architecture [4] or bounded contexts [5].

As software systems evolve, they also need to adapt their interfaces to changed functionality. A common way to inform clients calling these interfaces that their use is no longer encouraged is the *deprecation* of an entire API, an element in the API, or a particular version of the API. Depending on the further actions after a deprecation is introduced, it may lead to technical debt with both API clients [6] and API providers [7].

If clients wish to react to the deprecation of an API they depend on, they first need to be aware of the deprecation. While comprehensive support exists for detecting the deprecation of static APIs (e.g., for Java libraries) the situation is different for web APIs (e.g., for REST calls) [8].

Previous approaches for the detection of calls to deprecated web APIs have utilized static analysis methods similar to the case of static APIs [9]. However, this has two essential limitations: (i) The target of an API call can often not be determined statically. (ii) Deprecation in API specifications is not the only way to signal deprecation for web APIs. Thus, we introduce an approach to determine the usage of deprecated web APIs dynamically. To the best of our knowledge, it is the first such approach.

To overcome these limitations, we develop a new approach comprising two essential steps: (i) The execution of a client component is observed, and information about calls to web APIs is recorded. (ii) The recorded data is analyzed and each endpoint is checked for deprecation. Our approach considers deprecation that is signaled directly in the call's response, in an associated API specification, or in a knowledge base. We implement the approach for HTTP APIs and OpenAPI specifications. OpenAPI is a de-facto standard for the specification of REST APIs.

This paper's main contributions are:

- We present the first approach to identify the usage of deprecated web APIs dynamically.
- We implement the approach for REST APIs, OpenTelemetry data, and various forms of deprecation information.
- We evaluate the approach on multiple sample systems.
- We include a replication package with code, examples, and the evaluation data [10].

These contributions benefit practitioners who use or offer deprecated APIs. Furthermore, the contributions benefit researchers who wish to study the deprecation of web APIs.

Section II introduces fundamentals, motivates our research, and surveys related work. In Section IV, we introduce our approach and present its implementation. Subsequently, we evaluate the approach and its implementation in Section V and discuss our results and their limitations in Section VI. Finally, Section VII concludes the paper.

This is the accepted version of a research paper at the 21st IEEE International Conference on Software Architecture (ICSA 2024).

II. Background

In the following, we introduce the fundamentals relevant for our approach and the problem it tackles. In addition to scientific publications, we discuss relevant standards, as we will derive the design principles for our approach from them in the subsequent section.

A. API Evolution

APIs are interfaces that enable programmers to interact with other software components or systems by leveraging means of programming languages. This contrasts the interaction between systems based on data exchange files, binary interfaces, and manual interaction.

Although APIs are strongly coupled with the components or systems they are provided for, APIs themselves are also software artifacts. As such, they are often not only developed once but need to be maintained and evolve with changing requirements and contexts over time. Medjaoui et al. [11] model the lifecycle of APIs in five phases:

- **Create:** The API is in development and not available for clients yet. Deployed versions are considered prototypes, and locations, names, and behavior may still change frequently and unexpectedly.
- **Publish:** The API is deployed and available for at least one client. The API and its usage are monitored and if necessary, changes are made.
- **Realize:** The API is generally available and realizes business value. Stability is desired to avoid problems with client code, but changes are made to adapt the API to changed or new requirements or to fix errors.
- **Maintain:** The API is actively used, but no new features are introduced. Only necessary changes are performed, e.g., to secure a service against newly found security vulnerabilities.
- **Retire:** The API's end-of-life has been decided and announced. While it is still in operation, its usage is discouraged.

These API evolution activities introduce changes in new versions of an API. Depending on the nature of such a change, clients need to be adapted to the new version. Such API changes that would otherwise prevent the continued functioning of the API client are called *breaking-changes*, e.g., the removal of an existing method. Changes that do not force clients to react are called *non-breaking changes*, e.g., the addition of a new method. From the *Realize* stage onward, a *stable* API aims to avoid breaking changes.

In the *Retire* stage, to prepare clients for the upcoming removal of API elements, its usage may be discouraged by marking it as *deprecated*. In addition to upcoming removal, other reasons may also motivate a deprecation, e.g., another method provides better results in some cases.

B. API Types

Historically, several distinct types of APIs have emerged. The most important differentiation divides APIs into *local APIs* and *remote APIs* [12].

Local APIs are interfaces external to the software component under consideration, but reside within the same ecosystem and are governed by the syntactic rules of the programming language used. Local APIs may be provided by the programming language's SDK itself (e.g., a library to access operation system functionality) or through additional resources (e.g., a web framework). Research on local APIs has developed tool support for API providers and API clients, and has empirically investigated API evolution [13].

Local APIs are also called *static APIs*, *programming language APIs*, or *traditional APIs*: They are statically resolved [14], governed by the syntactic and semantic rules defined for the programming language used [15] and have been established prior to remote APIs [16].

In contrast, *remote APIs* use means of network communication (e.g., HTTP or WebSocket). They may additionally use message-oriented middleware (e.g., Apache Kafka), allowing for asynchronous interactions.

Web APIs are a subtype of remote APIs featuring synchronous request/response communication, e.g., REST, gRPC, and GraphQL [17]. While they have been originally developed in the context of the World Wide Web, they are also used in other contexts, e.g., microservice architectures [18]. For the purpose of this paper, no distinction between different technologies of web APIs is necessary and thus, we choose one of them, REST, for simplified illustration in the remainder.

Typical evolution of remote APIs has been described in several patterns [12], [19]–[21], e.g.,

- **Aggressive Obsolescence:** API providers enforce strict deadlines for the removal of old API versions. This forces API clients to react to the deprecation. Regarding the API lifecycle model by Medjaoui et al., this means keeping the *Retire* phase as short as possible.
- **Experimental Preview:** Unstable APIs and API versions are clearly communicated. In particular, preview versions (*Publish* phase) are subject to frequent change. Thus, deprecation is not necessary for such changes.
- **Limited Lifetime Guarantee:** API providers announce an API's end-of-life date with its introduction. This limits the *Realize* and the *Maintain* phases and allows for API clients to plan the required changes ahead of time.
- **Two in Production:** If a change to an API is introduced, the new version is published, but the old version is kept available as well. This allows clients to change from an API version in *Maintain* or *Retire* phase to an API in *Realize* stage deliberately.

While such patterns improve the management of API evolution, it remains a challenge to reliably inform clients of remote APIs about changes and prevail on them to adjust their code [21].

C. Deprecation of Static APIs

In the case of static APIs, deprecation is often an integral feature of a programming language and is generally well researched — in particular for the Java programming language and the Android ecosystem [8]. For Java, there is a comprehensive deprecation feature and a strong tool support [22]. In the Android ecosystem, swift reaction to the deprecation of an API in the Android SDK is essential, as a strict removal policy is enforced [23].

Java code is compiled into bytecode which is subsequently executed on a Java Virtual Machine. During compilation, the Java code is parsed, dependencies are resolved, and links to external resources, e.g., libraries, are created. If the compiler finds a called method marked as `@Deprecated`, it issues a warning. Even if a library later introduces a change to an API, the previously linked version is not affected as it is linked statically, i.e., stored in a local file.

D. Deprecation of Web APIs

In contrast to deprecation defined in programming language specifications, there is no standard way for the deprecation of web APIs. The information that an API is deprecated may be provided in a *non-technical* or in a *technical* way [24]. While non-technical communication is shared in natural language via an arbitrary channel (e.g., release notes, blog posts, or mailing lists), technical communication is presented in a machine-readable standardized format (e.g., API specification, header information in messages). An example for non-technical communication is the *API Deprecation Status* page by eBay [25].

In a closed ecosystem or if, for other reasons, all API clients are known, the API clients may be informed directly via non-technical channels. This additionally allows the API provider to reliably inspect who is still using the API and to support these clients in their migration to a new API version. However, a settings in which all customers are known are rare [26], [27]. For many API providers, their customers are unknown [21]. Thus, technical communication is necessary for API providers, and its reception is important for API clients.

1) Deprecation Information in API Specification: An API may be defined via an API specification describing the available operations and their intended usage, paths and names, and data types.

For REST APIs, the specification via *OpenAPI*[1] has become a de-facto standard after multiple competing providers of specification formats joined the OpenAPI Initiative under the Linux Foundation [21]. In version 3.0 of OpenAPI (released in 2017), a new field `deprecated` accepting a boolean value was added. This field is an optional addition to entire operations or single parameters of operation requests or responses. Alternatively, for OpenAPI versions prior to 3.0, deprecation may be expressed verbally in a description field (non-technical communication).

The specification for an API is often available through an API provider's web pages but often not linked directly from the API itself, i.e., provided within the communication with the API. The RFC 8631 standard defines such a header field (`Link` field with `service-desc` relation type) for API responses providing a link to the respective API specification [28].

An alternative is followed by another initiative that aims to standardize the URL path to API specifications via a "well-known" resource identifier [29]. The API specifications for a domain could then be discovered automatically under `/.well-known/api-catalog` [30].

2) Deprecation Information in Response Message: In the communication with an API, the response may contain (meta-)information in addition to the response itself, e.g., an HTTP status code. An API provider can communicate the deprecation of an API via this technical channel. However, there does not exist a standardized way for deprecation information in HTTP headers, but a variety of options:

A general `warning` header field was designed in 2014 for status information that cannot be expressed in HTTP status codes directly [31]. While this header field has been removed from the specification in 2022 [32], it is still used to signal deprecation, e.g., by Kubernetes [33].

The `sunset` header field communicates a date indicating the expected end-of-life of an API [34]. While this field is related to deprecation, it does not allow the exact same functionality as deprecation: Deprecation may not be limited to a certain date. In particular, an API may be deprecated, but not designated for removal at all. An additional `deprecation` header field has been in development [35], but has not been finalized yet. Nevertheless, it is used, e.g., by Zalando [36]. As a replacement, a more general `lifecycle` header field may combine the existing `sunset` and the planned `deprecation` [37].

Finally, non-standardized header fields are allowed in HTTP responses. Thus, some API providers define custom header fields, e.g., `X-API-Deprecation-Date` and `X-API-Deprecation-Info` by Zapier [38]. A standard demanded such custom header fields to begin with `X-`, but this limitation has been lifted [39].

3) Knowledge Base Providing Deprecation Information: While knowledge bases for technical aspects of commercial and open-source software exist, e.g., Technopedia[2], to the best of the authors' knowledge, no such knowledge base features machine-readable deprecation information on an API level, i.e., featuring individual operations that are deprecated. Such a knowledge base has been suggested in research literature [21], [24]. A knowledge base featuring the deprecation status of APIs can either be public, i.e., a repository representing general knowledge, or private, i.e., a project-specific collection of relevant APIs.

Although such a data source does not exist at the moment, we nevertheless designed our approach in a way that it could include such a service that checks a submitted URL for deprecation of the corresponding API.

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

[2]https://www.techopedia.com/

E. Observability and Tracing

Observability is the ability to collect telemetry data, i.e., data about the runtime behavior of software systems by collecting and exporting observation data [40]. Such data often comprise logs (i.e., textual event descriptions with corresponding timestamps), metrics (i.e., measurements of runtime properties), and traces (i.e., a series of state snapshots describing the execution of a single request throughout the system). Observability engineering is particularly concerned with distributed systems where tracing needs to follow a request through multiple services. Each service then generates partial traces (so-called *spans*) that need to be collected and combined into a full trace.

The *OpenTelemetry* framework[3] is an open-source project featuring specifications and tooling for vendor-agnostic observability. It accepts telemetry data in various formats and from various sources, and it exports the collected data to several backends. To collect telemetry data, the framework offers manual and automatic instrumentation of systems in major languages. For example, the automatic Java instrumentation inspects and injects bytecode that exports data whenever a system communicates with an external API. The collected telemetry data is subsequently passed to the *OpenTelemetry Collector*, which combines data from multiple sources, processes them (e.g., removes personal API keys from HTTP headers), and exports them to a backend service or logs them into a file.

III. Related Work

A. Analysis of Web API Deprecation

In 2022, Bonorden and Riebisch [8] systematically reviewed research publications on API deprecation. They found that scientific studies had been performed almost exclusively for local APIs. They proclaimed the deprecation of remote APIs to be "uncharted territory" since they could only identify a single study on the deprecation of REST APIs [24]. This confirms the results from Raatikainen et al. [14] who demand that research needs to study web APIs better to match the shift from local to remote APIs in industry.

Yasmin et al. [24] analyzed publicly available OpenAPI specifications regarding the use of deprecation fields. They found that 87% of API versions had not deprecated elements before they were removed. Furthermore, their data shows that request parameters have been deprecated more often (46%) than other elements of OpenAPI specifications. Finally, they studied the communication associated with deprecation: Some APIs do not share replacement information at all (33%) or only for some elements (22%); only 5% of deprecated APIs provide information about removal time. Only 1% of deprecated APIs provide the information additionally through response headers.

Di Lauro et al. [41] performed a similar analysis on a larger and more recent data set mined from GitHub data. They obtain similar results: APIs are often removed without deprecation (35%) or deprecated but not removed (64%). Only the remaining 2% of observed state changes followed the deprecate-remove protocol. Furthermore, they found that only a small set of APIs feature both a deprecation flag as well as deprecation information in the description field.

Lercher et al. [21] conducted 17 semi-structured interviews with developers, architects, and managers involved in maintaining APIs in systems featuring a microservice architecture. While they were focused on the API provider perspective, they nevertheless found that "developers do not think about the evolution of external APIs but expect unlimited availability of the consumed API version". They further found that OpenAPI is the de-facto standard to describe REST APIs, but half of the participants used additional manual documentation. They confirmed the use of the *Two in Production* pattern described by Zimmermann et al. [12]. Furthermore, all participants increment (major) API versions when introducing breaking changes, thus, keeping each individual version stable. Finally, they identified challenges for API evolution, two of which are related to deprecation: (1) If consumers rely on API functionality and do not prioritize updating to a newer version or are unable to do so, they still depend economically on the old API version. They conclude that API clients should be supported in migrating to newer API versions. (2) There often is no reliable way to inform all customers ahead of time about breaking changes. They conclude that tool support to detect usage of deprecated web APIs is necessary, e.g., by maintaining a list of actively used APIs and corresponding customers or by tracing actual calls still performed with the old API version.

B. Static Detection of Deprecation for Web APIs

Yasmin [9] implemented a static analysis approach to detect usage of deprecated REST APIs in Javascript software. They detect API requests by searching for common API call patterns and then working with backward slicing and data flow analysis to identify the strings passed into such calls. If they found such a URI string, they checked the deprecation status of the associated API in its OpenAPI specification. They evaluated the analysis with 432 client files that call 34 API operations from 2 API providers (Instagram and Github), yielding a precision of 100% and a recall of 77% for Instagram APIs and 94% for GitHub APIs, respectively.

Other studies follow different overall goals but include a step to identify calls to remote APIs in (Java) program code: Rapoport [42] searched for URL strings in Java code to locate calls to remote APIs but found that the statically identified calls and the additionally dynamically identified calls overlapped only slightly. Gadient [43] similarly tried to identify calls from Java code to remote APIs statically, but their approach yielded low precision (46%) and recall (80%). A method by Pigazzini et al. [44] focuses on a specific form of the remote call on the client side (i.e., they assume a specific form of calls using the *Feign* or *Spring RestTemplate* HTTP client), while the approach by Genfer & Zdun [45] assumes prior knowledge about the API called (i.e., a list of the available external endpoints in a microservice architecture).

[3]https://opentelemetry.io/

C. Use of Tracing Information to Study Web APIs

Tracing is a common practice to study web APIs. The main areas of application include performance analysis and architecture reconstruction. In performance analysis, traces are used to study runtime behavior and the fulfillment of related quality requirements, e.g., appropriate usage of system resources [46], [47]. Especially in systems featuring microservice architectures, traces are used to re-identify the composition of services and the communication between them [48], [49].

To the best of the authors' knowledge, tracing information has not been used yet to study the status of APIs calls, in particular, their deprecation status.

D. Assessment of the Current State

In conclusion, the analysis of related work on the detection of usage of deprecated web APIs reveals that there is only a static approach at the moment and that this approach has two essential limitations:

1) Call Target Identification: To statically detect usage of deprecated web APIs, information about the called endpoint needs to be extracted from source code alone. The endpoint can either be found directly (i.e., by searching for strings that likely resemble URLs) or indirectly (i.e., by identifying calls and reconstructing the parameters used). However, this approach is limited to scenarios in which the relevant information is actually contained in source code and does not rely on dynamically loaded external resources, e.g., user inputs, responses of prior communication, or file imports. Furthermore, for the indirect identification, the approach depends on the identification of HTTP calls, which are only a few options in some programming languages (e.g., Javascript with the built-in `XMLHttpRequest` and `Fetch` libraries), but a lot of possibilities in other programming languages (e.g., Java with little internal support for HTTP up to version 10, published in 2018, and, thus, many third-party libraries are in use).

2) Deprecation Information Gathering: In addition to the API call target itself, its deprecation status also needs to be identified. A static approach is limited to statically available data sources. Currently, the deprecation status is checked via API specifications, and a static approach could possibly be extended to include a knowledge base. However, the additional source of deprecation information, headers in call responses, cannot be accessed in static analysis as it is only produced upon dynamic requests.

In comparison with the deprecation of local APIs, static approaches for the deprecation of web APIs also feature another significant difference: While a local API is statically linked, a remote API might be deprecated at a later point in time. Thus, a static analysis at development or integration time is not sufficient to rule out deprecation for the entire lifetime of a component calling such an API. A static analysis would have to be updated continuously.

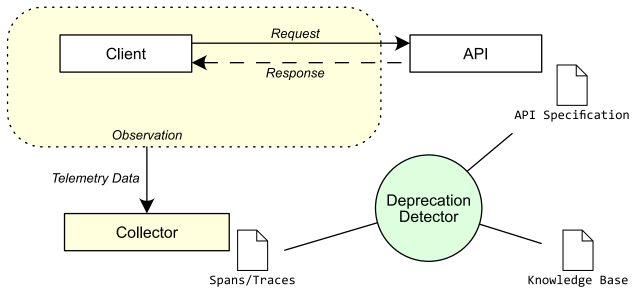

Fig. 1: Structural overview of the architecture for our approach to detect usage of deprecated web APIs.

IV. Approach

In this section, we present our approach and its sample implementation used for the evaluation. We differentiate between the approach's *design* and its *implementation*. Intentionally, the design does not dependent on a specific implementation technology, but may be applied broadly for various kinds of protocols and data structures of APIs, their specifications, and observation data.

A. Design

1) Goals and Design Principles: Analyzing related literature, we formulate a goal and desired design principles for our approach. The overall goal is **to identify calls from clients to deprecated web APIs** and we formulate the accompanying design principles:

DP1 The identification should not be targeted to a single programming language.

DP2 The identification should include calls with targets that are only known at runtime.

DP3 The identification should include deprecation information from API specifications, HTTP header data in responses, and knowledge bases.

DP4 No prior information about an API should be required for the identification.

2) Overview: Our approach comprises two essentially distinct parts: An *observation* to record runtime data about calls to web APIs and a *detection* to match the runtime data with deprecation information about the called APIs. A graphical overview of this approach is given in Fig. 1.

We point out that the API called is neither part of the observation nor the detection. Only its response and its associated specification are used. No additional calls to the API are generated by our approach.

3) Observation: The observation records runtime data. We are interested in the following information about calls to web APIs:

- the call target, i.e., a URL:
  `https://example.com/api/v2/resource/5`

- the HTTP method used, i.e., one of the five pre-defined HTTP verbs:
  `GET`
- the header fields of the response, i.e., a collection of possibly many key-value pairs:
  ```
  Sunset: Tue, 31 Dec 2024 23:59:59 GMT
  Link: <https://example.com/api-docs>;
      rel="service-desc"
  ```

The URL and the HTTP method are mandatory elements to perform a deprecation analysis. The URL is the essential information to decide which API element is called and needs to be checked for deprecation. Additionally, the HTTP method is further necessary to identify the exact operation as one element may define a separate operation for any of the HTTP methods. Finally, the fields in the response header are needed, if a deprecation is indicated this way and should be checked, or if the path to the API specification should be obtained from header information. If the deprecation status should only be checked via an already known API specification or a knowledge base, this information does not need to be recorded.

The URL may be given in parts (e.g., the scheme `https://`, the host `example.com` and the path `api/v2/resource/5`), or contain additional information (e.g., a query `?k=example`, or a fragment `#part`).

Such an observation typically comprises two parts: the client instrumentation per se, and a collection of the observed data. This setup allows for the client and its instrumentation to be deployed elsewhere than the data collection. In particular, multiple clients can provide information that is collected in a single place, e.g., multiple services in a microservice architecture.

Nevertheless, the approach is not limited to this two-part scenario. A suitable client could write its own data directly into a file accessible to the detector.

4) Detection: The detection uses the recorded runtime data from the observation to check the called APIs for deprecation via multiple sources of deprecation information.

First, deprecation via header information is checked. For this, a set of pre-defined header fields is checked. If one of the fields `Sunset` or `Deprecate` is present, the deprecation is certain. If a `Lifecycle` element is detected, it needs to be checked if a deprecation-related status (i.e., `deprecation=@2024-12-31T23:59:00Z` or `sunset=@2025-12-31T23:59:00Z`). The list could be extended for custom elements, e.g., `X-API-Deprecation-Info`.

Furthermore, the `Link` field is checked: If a link with relation type `service-desc` is found, the linked API specification is accessed. If it actually is a readable API specification, it is used for the analysis in subsequent steps.

Second, the matching with external resources, i.e., API specifications or a knowledge base, is prepared. In this step, the exact API endpoint that has been called by the client needs to be determined. If the URL is given in parts, it is combined into a single URL. If the URL contains additional information (e.g., a query) it is removed. As the URL contains both, the server's base URL (i.e., the scope of a single API specification) and a path (a concrete API element), it is not possible to directly extract the distinct parts, but the path is considered as one element. For example, the URL `https://example.com/api/v2/resource/5` could be

```
Base: https://example.com/api/v2
Path: /resource/5
```

or

```
Base: https://example.com/api
Path: /v2/resource/5
```

depending on the positioning and scope of the corresponding API specification.

Third, the URL is matched and checked against external resources. For API specifications, the base URL specified in the specification file is checked against the recorded target URL. The comparison is a match, if the specified URL is found at the beginning of the recorded URL – possibly followed by additional elements. For a knowledge base, either the entire URL could be supplied or only the host. This depends on the design of the knowledge base as either it executes the check with a complete URL, or constitutes a repository of API specifications for host URLs. If multiple API specifications for one host are returned, the detection needs to check all of them for a matching base URL.

Finally, the detection publishes its result. There are a variety of ways for this step, depending on the intended workflow for warnings about the usage of deprecated elements. For example, the information could be logged into a file and analyzed later, or the information could be used to automatically create an issue in a project management software.

B. Implementation

For the implementation, we decided to use REST APIs and OpenAPI specifications as these are among the most used variations [21]. Furthermore, we use OpenTelemetry as it is an open standard compatible with various vendor-specific formats and tools. The implementation is provided in the replication package [10].

1) Observation/Instrumentation: The observation is performed by the automatic client instrumentation provided by OpenTelemetry. This part of the OpenTelemetry framework inspects code or bytecode, searches for known ways to perform external calls, and injects the necessary information to record and extract data. We customize the automatic instrumentation explicitly demanding the recording of the necessary header fields:

```
otel.instrumentation.http.client.
    capture-response-headers=
    "sunset,deprecation,lifecycle,link"
```

Furthermore, we provide the address for the collector and state the desired data formats.

Listing 1: Configuration for the OpenTelemetry Collector

```
exporters:
  file:
    path: ./output.json
service:
  pipelines:
    traces:
      exporters: [file]
```

Listing 2: Partial attribute section in a recorded HTTP span

```
"attributes":[
 {
  "key":"http.url",
  "value":{"stringValue":"https://example
     .com/api/v2/resource/5"}
 },
 {
  "key":"http.response.header.sunset",
  "value":{"arrayValue":{"values":[{"
     stringValue":"Tue, 31 Dec 2024
     23:59:59 GMT"}]}}
 },
 {
  "key":"http.method",
  "value":{"stringValue":"GET"}
 }
]
```

2) Collection: The recorded runtime data is forwarded to the collector. For this, we use the OpenTelemetry Collector. While the OpenTelemetry Collector allows simple preprocessing of data and export to a variety of monitoring backends, we are mostly interested in the raw spans or traces collected. To preserve the independence from other tooling, we simply export the relevant data to a file using the configuration shown in Listing 1. The exported file contains HTTP spans for all recorded calls to APIs. Listing 2 shows an example of the attribute section in such a span recording.

3) Detection: For the analysis of the header information, the detection reads the output file exported by the OpenTelemetry Collector. Subsequently, it checks known API specifications, i.e., OpenAPI files in a local folder. Additionally, it may consult a knowledge base via a known address. This knowledge base is implemented for demonstration purposes and features a single operation `/deprecated` accepting a URL via a POST request and returning a boolean value to indicate the deprecation of the provided URL. An example of a deprecation flag in an OpenAPI specification is shown in Listing 3.

Finally, the detection exports its results. For simplification in the sample implementation, this is currently a basic JSON file export. An example for a result is shown in Listing 4.

Listing 3: Excerpt from an OpenAPI specification featuring a deprecated operation

```
"/resource/{resourceId}": {
    "get": {
        "summary": "Get information about
             a specific resource",
        "description": "...",
        "deprecated": true,
        "parameters": ... ,
        "produces": "application/json"
        "responses": ...
    }
}
```

Listing 4: Result for an API that has been deprecated via response headers and in an API specification

```
{
  "URL": "https://example.com/api/v2/
     resource/5",
  "httpMethod": "GET",
  "apiSpecification": "example-api.json",
  "deprecated": true,
  "deprecationInformation": [
    "Sunset field in response header: Tue
       , 31 Dec 2024 23:59:59 GMT",
    "Operation deprecated in API
       specification: specifications/
       example-api.json"
  ]
}
```

V. Evaluation

To evaluate our approach and its implementation empirically, we apply it to multiple sample systems and ask the following research questions about precision and recall:

RQ1 How *precise* is our approach in detecting calls to deprecated web APIs?

RQ2 How *sensitive* is our approach in detecting calls to deprecated web APIs?

A. Method

To answer the research questions, we apply the implementation of our approach to multiple sample systems and compare the findings to the results of a manual inspection. A direct comparison with other approaches would have been desirable, but unfortunately, it is not possible: To the best of our knowledge, no other dynamic approach has been reported. The only other approach to this problem is the static identification by Yasmin [9]. However, no tool or data for their research is available.

We have selected the following projects to be used in the evaluation and included them in the replication package:

1) Petstore: This project is an established example for testing OpenAPI specifications featuring a pet store management system. A specification file and a corresponding server [4] are available. The project features multiple API endpoints and some of these endpoints are available for multiple HTTP methods. Thus, we can evaluate the exact matching of operations via path and HTTP method.

We deprecated the `GET /pet/{petId}` operation in the OpenAPI specification and stored the modified specification file locally with our implementation. We then recorded calls from a client application that performed various calls to the Petstore API including multiple calls to `/pet/{petId}` using various HTTP methods.

We manually inspected which client calls use the deprecated API and compared our detection results against the manual results.

2) Lakeside Mutual: This project is a fictitious case study of an insurance company described in [12] and released as open-source software[5]. The system has been designed following the Domain-Driven Design method and is implemented in a microservice architecture. It features nine microservices written in Java and JavaScript and two administrative services.

We selected one of the microservices, the *Customer Core*, and marked all of its APIs as deprecated. To implement this modification, we added `sunset` and `deprecation` HTTP response headers. Additionally, we added a log statement to each API for later comparison.

Subsequently, we setup our approach's implementation and collected runtime data while performing a manual test sequence in the system's web frontends. Afterwards, we ran the deprecation detection on the collected data. We compared the results with a manual inspection and the log data from the modified microservice.

3) client-server: In addition to these projects, our replication package features a basic client-server application that includes the features not covered by the selected applications. This application comprises a Spring Boot server offering multiple API paths with different deprecation statuses and different information channels used to communicate the deprecation:

- deprecation information in response header fields, one operation for each of: `Sunset`, `Deprecation`, `Lifecycle`,
- deprecation information in an OpenAPI specification that is linked in a `Link` header field in the response,
- no deprecation information in response header fields or an API specification, but a knowledge base featuring deprecation information.

The corresponding client application interacts with all of the server's endpoints. Again, we manually inspected the client and compared the results from the automatic analysis against the manual findings.

[4]https://petstore3.swagger.io/

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

TABLE I: Evaluation results

Project	Deprecation via	Precision	Recall
Petstore	OpenAPI specification	1.00	0.75
Lakeside Mutual	HTTP headers	1.00	1.00
client-server	HTTP headers, OpenAPI specification, knowledge base	1.00	1.00
Total		1.00	0.95

B. Results

1) Petstore: For the Petstore project, we achieved a precision of 1.00, but could only observe a recall of 0.75 due to some false negatives. We investigated the situation in detail and found that the project uses two different clients to perform HTTP calls: the *HttpClient* provided directly by Java, and the *Apache HttpClient 5.2*. While the communication performed with the HttpClient was correctly observed, the communication using the Apache HttpClient 5.2 was not.

The Apache HttpClient 5.2 features multiple ways to call HTTP APIs, in particular a "fluent" style. This style is currently not covered by the automatic instrumentation provided by OpenTelemetry. Thus, we could not record any spans for these calls and, subsequently, could not analyze this communication.

2) Lakeside Mutual: The analysis for the Lakeside Mutual project provides results that match the manual analysis exactly. Thus, precision and recall of 1.00 are reached.

3) client-server: Similar to the Lakeside Mutual project, we also achieved precision and recall of 1.00 for the client-server project as the results from manual and automatic analysis match perfectly.

4) Overall results: An overview for the results of each project is given in Table I. This allows us to answer the research questions for the empirical evaluation:

RQ1: How *precise* is our approach in detecting calls to deprecated web APIs? The implementation for our approach yields a precision of 1.00 for the example projects. This signifies that all results reported are actually relevant and correspond to calls to deprecated web APIs.

RQ2: How *sensitive* is our approach in detecting calls to deprecated web APIs? The implementation for our approach yields a recall of 0.95 for the example projects. The false negatives can be attributed to a shortcoming in the automatic instrumentation for Java provided by OpenTelemetry. This implies that the approach might miss a call to a deprecated web API if the instrumentation is not performed correctly.

C. Threats to Validity

In the following, we discuss threats to validity of the evaluation. The limitations of the approach itself will be discussed in the subsequent section.

Representation bias: The projects used in the empirical evaluation may not be representative – in particular the evaluation is limited to very few projects. Thus, the generalizability may be limited which poses a threat to external validity. We mitigated this threat by the addition of the sample project to cover all cases that have been described in standards and scientific literature.

Experimenter bias: We have established the ground truth for comparison by manual analysis. Thus, the correctness depends on the experimenters and their understanding and their flawless execution of the detection task. This poses a threat to construct validity. We mitigated this threat by additionally performing static analysis to help the manual analysis. For this, we searched the source code for strings that could resemble URLs (e.g., starting with `http` or `https`) and for method invocations of common HTTP libraries (e.g., calls to `execute` methods in Java classes that feature an `import okhttp3` statement).

VI. Discussion

The evaluation shows that it is possible to detect calls to deprecated web APIs dynamically by using tracing. In this section, we further discuss our approach, the implications of our findings, and the approach's limitations.

A. Adherence to Design Principles

We find that our approach fulfills all of the stated design principles:

DP1: The identification should not be targeted to a single programming language. While the instrumentation is specific for each programming language, the approach does not depend on a specific instrumentation or programming language. Even programming languages for which no automatic instrumentation exists at the moment may export the desired information manually. Our replication package demonstrates the instrumentation for Java and Python.

DP2: The identification should include calls with targets that are only known at runtime. Our approach captures runtime information for all calls to web APIs and does not require static information that would have been extracted from source code. Our replication package contains an example where a first HTTP call is used to obtain the URL for a subsequent second HTTP call that is unknown prior to the execution.

DP3: The identification should include deprecation information from API specifications, HTTP header data in responses, and knowledge bases. Our approach uses all three sources for deprecation information. The implementation and the sample projects in the replication package demonstrate this capability.

DP4: No prior information about an API should be required for the identification. No prior information about the API that is called by the client application is needed. The API or its specification do not need to be known apriori. However, our approach does support the prior collection of API specifications.

B. Recommendations

We found that it is possible to detect clients using deprecated web APIs. However, this detection requires API providers to systematically use technical information channels to communicate the deprecation of web APIs.

This two-fold requirement could pose a "chicken or the egg" problem: API providers might delay the implementation of these communication channels if clients do not use the detection possibilities and request information from API providers. Contrarily, API clients might refrain from checking the deprecation status of the APIs they call if API providers do not provide their information.

To overcome this situation, we formulate recommendations for both, API providers and API clients:

Recommendation for API Providers

- If an API provider decides to deprecate an API, they should use a technical communication channel or both, technical and non-technical communication channels.
- Deprecation should be added in both, response headers and API specifications.

Recommendation for API Clients

- API clients should review header fields related to deprecation at runtime.
- API clients should consult API specifications to find deprecation-related information.
- Both processes may be done manually/periodically, but executing them automatically/continuously should be preferred.

C. Limitations

In addition to the threats to the validity of the empirical evaluation discussed in the previous section, the approach itself is subject to some limitations:

Observability: The approach requires the availability of HTTP spans or complete traces. These can only be collected by instrumentation of the client code, thus depending on the source code's availability. If an application or part of an application is only available in binary format or as a container, it cannot be instrumented. Thus, the approach is a white-box technique.

Overhead: Tracing introduces additional operations which adds an overhead to runtime, memory usage, and initialization duration [50]–[52]. This overhead is negligible if tracing has already been used and only the deprecation detection is added, but particularly concerning if tracing needs to be introduced in the first place.

Instrumentation: If automatic instrumentation of client code is used, the approach depends on the correctness of this instrumentation. The evaluation has shown that some call may be missed if a client is used that is not covered by the automatic instrumentation. If manual instrumentation is used, the success of the analysis depends on the correct manual implementation.

Deprecation information: Another requirement is the availability of machine-readable deprecation information. If a deprecation is only communicated via non-technical channels, the detection cannot be successful. However, the approach might be complemented by a transformer from non-technical to technical information.

Manual maintenance: Since multiple options to deprecate an API in HTTP response header fields exist, the successful application of our approach depends on the coverage of such options. As long as such options are not standardized and the standard is adopted by all participants, it is necessary to manually maintain a list of HTTP header fields that need to be observed in responses from APIs.

Coverage: Since the approach utilizes dynamic data collected at runtime, it can only cover communication with APIs that is actually performed in an observed session. Thus, similar to tests, the quality of the overall results heavily depends on the coverage achieved during the execution. Furthermore, this dynamic approach should be complemented with other (static) approaches.

D. Future Work

While we have evaluated our approach positively, we nevertheless observe opportunities for subsequent future investigations:

Applicability: The approach is designed to assist humans, i.e., developers of API clients. Thus, the actual usefulness of the approach needs to be evaluated with an experiment involving human participants.

Presentation of warnings: Currently, our sample implementation exports the findings into a file. The best way to process and present such information needs to be examined. This includes the possible filtering of results to avoid alarm fatigue, i.e., ignoring warnings due to a high number of simultaneous alarms [27].

Industrial application: The approach should be evaluated in a case study or using action research in a real organization. In particular, the approach should not be implemented on its own, but as part of a larger context. This would further improve the external validity of our study.

Hybrid approach: Our approach inherits the general downside of dynamic approaches, in particular the limited coverage. A future study should combine our approach with the static detection of deprecated APIs to constitute a balanced hybrid approach.

Dynamic black-box approach: It should be researched if dynamic data could also be collected if direct client instrumentation is not possible, e.g., with the analysis of network traffic via an HTTPS proxy server.

VII. Conclusion

In this paper, we examined the detection of client calls to deprecated web APIs. Client developers need to be aware that they are using deprecated APIs to be able to react to the deprecation, e.g., by updating to a new version. This is substantially more complex for web APIs than it is for local APIs.

To design our approach, we studied related work and relevant standards to aggregate requirements. Subsequently, we designed a dynamic strategy to record runtime data from clients and analyze it. The analysis considers three important sources of information about the deprecation status of an API: (i) header fields in the HTTP response, (ii) deprecation flags in an API specification, and (iii) knowledge bases.

We implemented our approach and evaluated it on three projects. The evaluation achieves a precision of 1.00 and a recall of 0.95. Thus, we conclude that our approach achieves good results while simultaneously overcoming the limitations of static approaches.

Since our dynamic approach introduces new limitations in comparison to a static approach, it will be important to advance both strategies and possibly combine them in future work.

References

[1] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.

[2] N. A. Ernst, I. Ozkaya, R. L. Nord, J. Delange, S. Bellomo, and I. Gorton, "Understanding the Role of Constraints on Architecturally Significant Requirements," in *3rd International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks)*, 2013, pp. 9–14. [Online]. Available: https://doi.org/10.1109/TwinPeaks-2.2013.6617353

[3] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th ed. Pearson, 2021.

[4] M. Richards and N. Ford, *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly, 2020.

[5] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Pearson, 2015.

[6] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt," in *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015, p. 50–60. [Online]. Available: https://doi.org/10.1145/2786805.2786848

[7] A. Sundelin, J. Gonzalez-Huerta, and K. Wnuk, "The Hidden Cost of Backward Compatibility: When Deprecation Turns into Technical Debt - an Experience Report," in *3rd International Conference on Technical Debt (TechDebt)*, 2020, p. 67–76. [Online]. Available: https://doi.org/10.1145/3387906.3388629

[8] L. Bonorden and M. Riebisch, "API Deprecation: A Systematic Mapping Study," in *48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2022, pp. 451–458. [Online]. Available: https://doi.org/10.1109/SEAA56994.2022.00076

[9] J. Yasmin, "RESTful API Deprecation: An Empirical Study on Deprecation Practices and a Method for Detecting Deprecated API Usages in Client-side Web Applications," Master's thesis, Queen's University, Kingston, Ontario, Canada, 2021.

[10] L. Bonorden and A. von Hoorn, "Detecting Usage of Deprecated Web APIs via Tracing: Replication Package," Zenodo, 2023. [Online]. Available: https://doi.org/10.5281/zenodo.10250357

[11] M. Medjaoui, E. Wilde, R. Mitra, and M. Amundsen, *Continuous API Management*, 2nd ed. O'Reilly, 2021.

[12] O. Zimmermann, M. Stocker, D. Lübke, U. Zdun, and C. Pautasso, *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley Professional, 2022.

[13] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang, "A Systematic Review of API Evolution Literature," *ACM Computing Surveys*, vol. 54, no. 8, 2021. [Online]. Available: https://doi.org/10.1145/3470133

[14] M. Raatikainen, E. Kettunen, A. Salonen, M. Komssi, T. Mikkonen, and T. Lehtonen, "State of the Practice in Application Programming Interfaces (APIs): A Case Study," in *15th European Conference on Software Architecture (ECSA)*, 2021, pp. 191–206. [Online]. Available: https://doi.org/10.1007/978-3-030-86044-8_14

[15] L. Liu, M. Bahrami, J. Park, and W.-P. Chen, "Web API Search: Discover Web API and Its Endpoint with Natural Language Queries," in *IEEE International Conference on Web Services (ICWS)*, 2020, pp. 96–113. [Online]. Available: https://doi.org/10.1007/978-3-030-59618-7_7

[16] E. Wittern, A. T. Ying, Y. Zheng, J. A. Laredo, J. Dolby, C. C. Young, and A. A. Slominski, "Opportunities in software engineering research for web api consumption," in *IEEE/ACM 1st International Workshop on API Usage and Evolution (WAPI)*, 2017, pp. 7–10. [Online]. Available: https://doi.org/10.1109/WAPI.2017.1

[17] A. Lauret, *The Design of Web APIs*. Manning, 2019.

[18] A. Bellemare, *Building Event-Driven Microservices*. O'Reilly, 2020.

[19] D. Lübke, O. Zimmermann, C. Pautasso, U. Zdun, and M. Stocker, "Interface Evolution Patterns: Balancing Compatibility and Extensibility across Service Life Cycles," in *24th European Conference on Pattern Languages of Programs (EuroPlop)*, 2019. [Online]. Available: https://doi.org/10.1145/3361149.3361164

[20] O. Zimmermann, M. Stocker, D. Lübke, C. Pautasso, and U. Zdun, "Introduction to Microservice API Patterns (MAP)," in *First and Second International Conference on Microservices (Microservices 2017/2019)*, 2020, pp. 4:1–4:17. [Online]. Available: https://doi.org/10.4230/OASIcs.Microservices.2017-2019.4

[21] A. Lercher, J. Glock, C. Macho, and M. Pinzger, "Microservice API Evolution in Practice: A Study on Strategies and Challenges," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2311.08175

[22] A. A. Sawant, M. Aniche, A. van Deursen, and A. Bacchelli, "Understanding developers' needs on deprecation as a language feature," in *IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 561–571. [Online]. Available: https://doi.org/10.1145/3180155.3180170

[23] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "CDA: Characterising Deprecated Android APIs," *Empirical Software Engineering*, vol. 25, pp. 2058–2098, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09764-z

[24] J. Yasmin, Y. Tian, and J. Yang, "A First Look at the Deprecation of RESTful APIs: An Empirical Study," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 151–161.

[25] eBay, "API Deprecation Status," https://developer.ebay.com/develop/apis/api-deprecation-status, 2023, archived snapshot: https://web.archive.org/web/20230803115701/https://developer.ebay.com/develop/apis/api-deprecation-status.

[26] J. D. Morgenthaler, "Deprecation," in *Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)*, 2016, p. 134. [Online]. Available: https://doi.org/10.4230/DagRep.6.4.110

[27] T. Winters, T. Manshreck, and H. Wright, *Software Engineering at Google*. O'Reilly, 2020.

[28] E. Wilde, "Link Relation Types for Web Services," RFC 8631, 2019. [Online]. Available: https://www.rfc-editor.org/info/rfc8631

[29] M. Nottingham, "Well-Known Uniform Resource Identifiers (URIs)," RFC 8615, 2019. [Online]. Available: https://www.rfc-editor.org/info/rfc8615

[30] K. Smith, "A well-known URI to help discovery of APIs," IETF, Internet-Draft, 2023, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-httpapi-api-catalog/00/

[31] R. T. Fielding, M. Nottingham, and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Caching," RFC 7234, 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7234

[32] ——, "HTTP Caching," RFC 9111, 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9111

[33] Kubernetes, "Kubernetes Deprecation Policy," https://kubernetes.io/docs/reference/using-api/deprecation-policy/, 2023, archived snapshot: https://web.archive.org/web/20230822152543/https://kubernetes.io/docs/reference/using-api/deprecation-policy/.

[34] E. Wilde, "The Sunset HTTP Header Field," RFC 8594, 2019. [Online]. Available: https://www.rfc-editor.org/info/rfc8594

[35] S. Dalal and E. Wilde, "The Deprecation HTTP Header Field," IETF, Internet-Draft, 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-httpapi-deprecation-header/02/

[36] Zalando, "API Guidelines," https://opensource.zalando.com/restful-api-guidelines/, 2021, archived snapshot: https://web.archive.org/web/20231115111747/https://opensource.zalando.com/restful-api-guidelines/.

[37] E. Wilde, "from "Deprecation" to "Lifecycle": asking for feedback," https://mailarchive.ietf.org/arch/msg/httpapi/cI8Ejxs8ES_vJvxOQvtU0jE1_eY/#, message to the IETF [httpapi] mailing list.

[38] Zapier, "API Lifecycle, Versioning, and Deprecation," https://zapier.com/engineering/api-geriatrics/, 2017, archived snapshot: https://web.archive.org/web/20230318233502/https://zapier.com/engineering/api-geriatrics/.

[39] P. Saint-Andre, D. Crocker, and M. Nottingham, "Deprecating the "X-" Prefix and Similar Constructs in Application Protocols," RFC 6648, 2012. [Online]. Available: https://www.rfc-editor.org/info/rfc6648

[40] C. Majors, L. Fong-Jones, and G. Miranda, *Observability Engineering*. O'Reilly, 2022.

[41] F. Di Lauro, S. Serbout, and C. Pautasso, "To Deprecate or to Simply Drop Operations? An Empirical Study on the Evolution of a Large OpenAPI Collection," in *16th European Conference on Software Architecture (ECSA)*, 2022, pp. 38–46. [Online]. Available: https://doi.org/10.1007/978-3-031-16697-6_3

[42] M. Rapoport, P. Suter, E. Wittern, O. Lhotak, and J. Dolby, "Who you gonna call? analyzing web requests in android applications," in *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 80–90. [Online]. Available: https://doi.org/10.1109/MSR.2017.11

[43] P. Gadient, M. Ghafari, M.-A. Tarnutzer, and O. Nierstrasz, "Web apis in android through the lens of security," in *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 13–22. [Online]. Available: https://doi.org/10.1109/SANER48275.2020.9054850

[44] I. Pigazzini, F. A. Fontana, V. Lenarduzzi, and D. Taibi, "Towards Microservice Smells Detection," in *3rd International Conference on Technical Debt (TechDebt)*, 2020, p. 92–97. [Online]. Available: https://doi.org/10.1145/3387906.3388625

[45] P. Genfer and U. Zdun, "Identifying Domain-Based Cyclic Dependencies in Microservice APIs Using Source Code Detectors," in *15th European Conference on Software Architecture (ECSA)*, 2021, pp. 207–222. [Online]. Available: https://doi.org/10.1007/978-3-030-86044-8_15

[46] A. van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *3rd ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2012, p. 247–248. [Online]. Available: https://doi.org/10.1145/2188286.2188326

[47] C. Cassé, P. Berthou, P. Owezarski, and S. Josset, "A tracing based model to identify bottlenecks in physically distributed applications," in *2022 International Conference on Information Networking (ICOIN)*, 2022, pp. 226–231. [Online]. Available: https://doi.org/10.1109/ICOIN53446.2022.9687217

[48] B. Mayer and R. Weinreich, "An approach to extract the architecture of microservice-based software systems," in *IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2018, pp. 21–30. [Online]. Available: https://doi.org/10.1109/SOSE.2018.00012

[49] S. Singh, D. Werle, and A. Koziolek, "Archi4mom: Using tracing information to extract the architecture of microservice-based systems from message-oriented middleware," in *European Conference on Software Architecture (ECSA)*, 2022, pp. 189–204. [Online]. Available: https://doi.org/10.1007/978-3-031-16697-6_14

[50] D. G. Reichelt, S. Kühne, and W. Hasselbring, "Overhead comparison of opentelemetry, inspectit and kieker," in *Symposium on Software Performance 2021*, ser. CEUR Worshop Proceedings, 2021. [Online]. Available: http://ceur-ws.org/Vol-3043/

[51] D. G. Reichelt, S. Kühne, and W. Hasselbring, "Towards solving the challenge of minimal overhead monitoring," in *Companion of the 14th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2023, p. 381–388. [Online]. Available: https://doi.org/10.1145/3578245.3584851

[52] C. Eder, S. Winzinger, and R. Lichtenthäler, "A comparison of distributed tracing tools in serverless applications," in *17th IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2023, pp. 98–105. [Online]. Available: https://doi.org/10.1109/SOSE58276.2023.00018