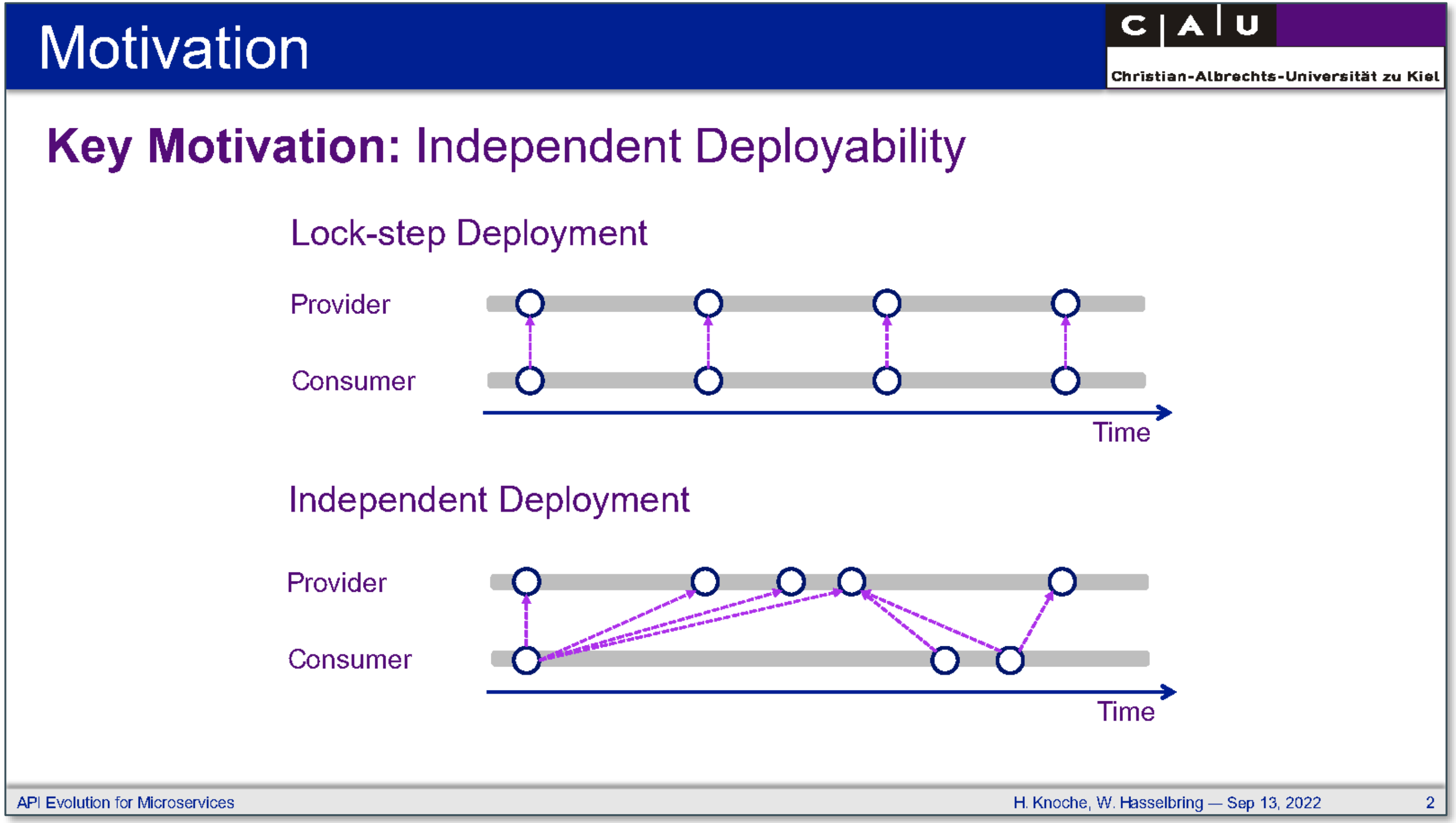
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Detecting Usage of Deprecated Remote APIs

April 04, 2024 **Leif Bonorden** – leif.bonorden@uni-hamburg.de

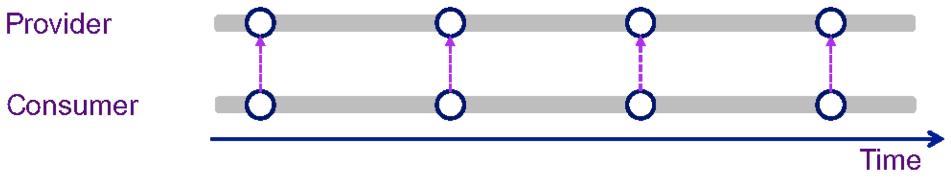


Motivation

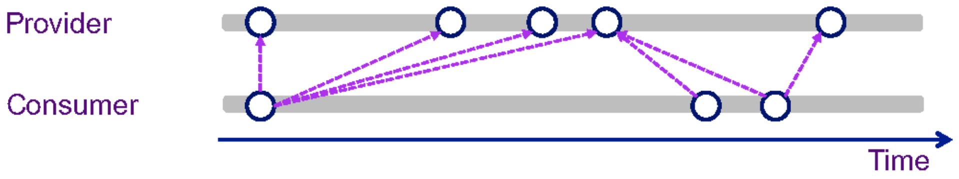
Christian-Albrechts-Universität zu Kiel

Key Motivation: Independent Deployability

Lock-step Deployment

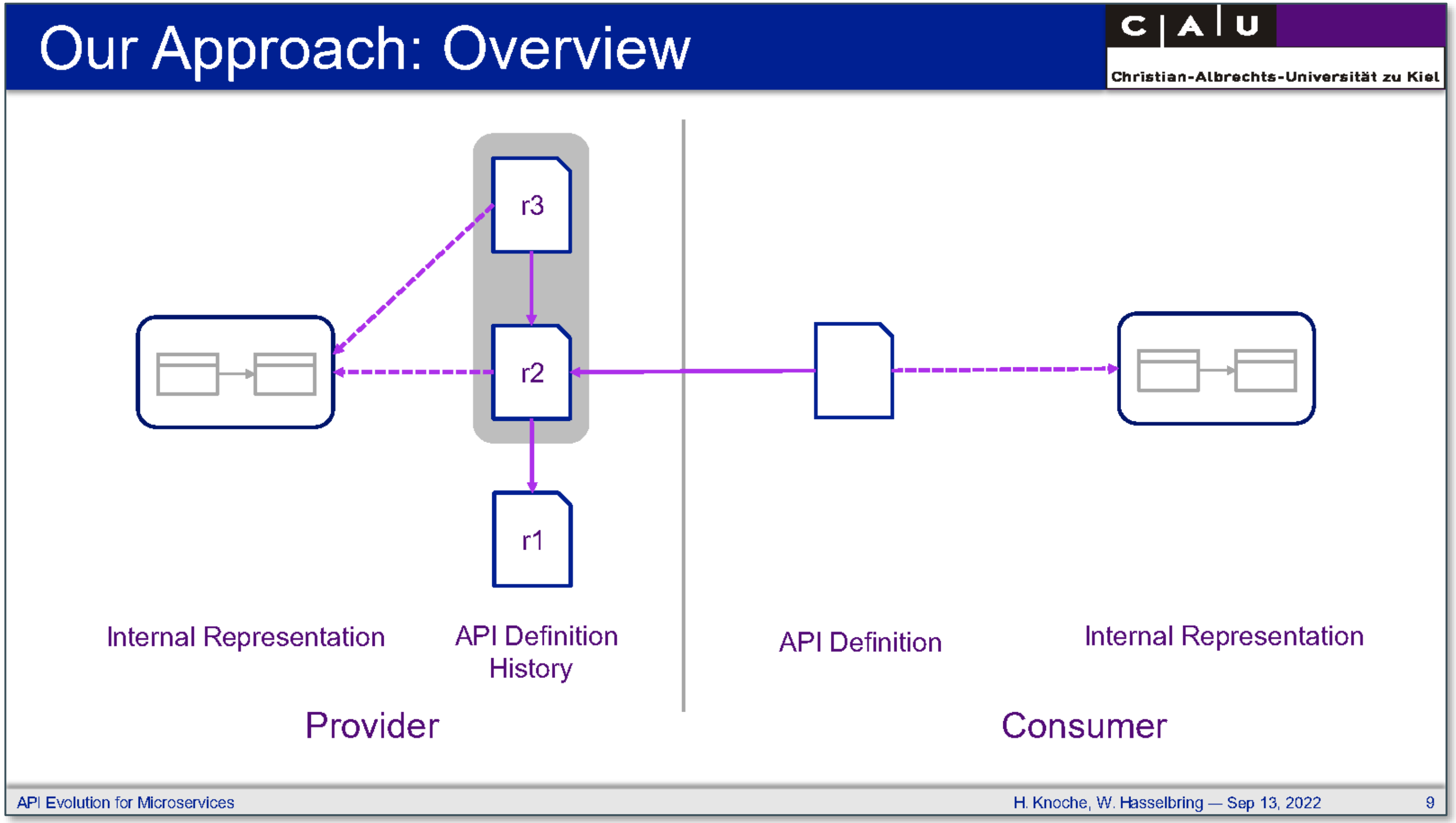


Independent Deployment



API Evolution for Microservices H. Knoche, W. Hasselbring — Sep 13, 2022 2

Holger Knoche: *API Evolution for Microservices*.
8. Treffen des AK MSDO, 2022.



Our Approach: Overview

Christian-Albrechts-Universität zu Kiel

API Evolution for Microservices H. Knoche, W. Hasselbring — Sep 13, 2022 9

Holger Knoche: *API Evolution for Microservices*. 8. Treffen des AK MSDO, 2022.

Breaking Changes

changes in an API that potentially break existing client code

– violation of backward compatibility

Examples

- removing a method
- renaming an element

Counter Examples (non-breaking)

- adding a new method
- increasing an element's visibility

Deprecation

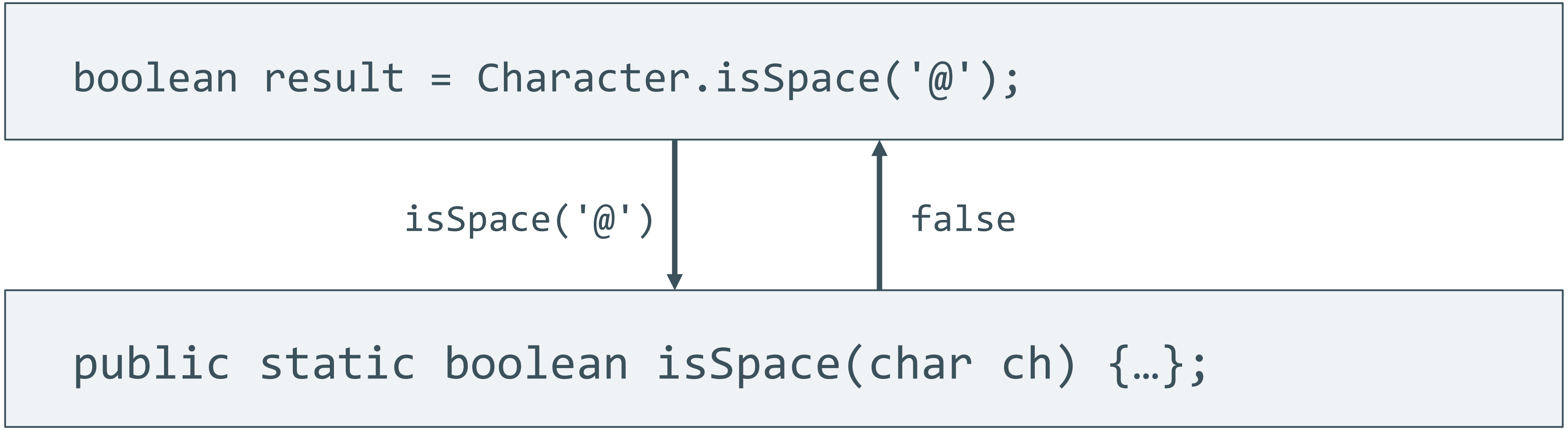
deprecated

the use of this element is discouraged

Typical reasons

- the element will be removed/changed (→ breaking changes)
- a better alternative exists (e.g., faster)
- the element is not reliable (e.g., not thread-safe, non-deterministic)

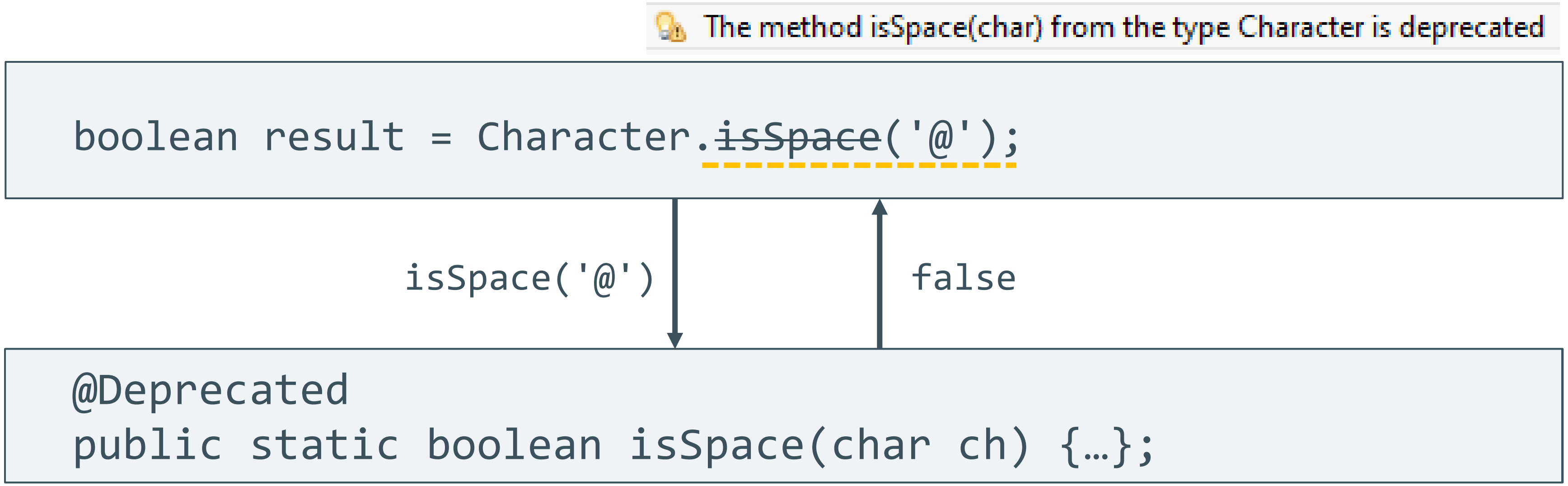
Deprecation: Java APIs



```
boolean result = Character.isSpace('@');
```

```
public static boolean isSpace(char ch) {…};
```

Deprecation: Java APIs


The method isSpace(char) from the type Character is deprecated

```
boolean result = Character.~~isSpace~~('@');
```

```
@Deprecated
public static boolean isSpace(char ch) {…};
```

Deprecation: remote APIs

```
HttpClient cl = HttpClient.newBuilder().build();

HttpRequest req = HttpRequest.newBuilder().
                       uri("https://character.com/isSpace").build();

HttpResponse res = client.send(request, myBodyHandler);
```

```
POST /isSpace HTTP 1.1
   Host: character.com

{ … "character":"@", …}
```

```
HTTP/1.1 200 OK
Date: Thu, 04 Apr 2024

{ … "result":false, … }
```

OPENAPI INITIATIVE

Deprecation: remote APIs

The API element *isSpace* is deprecated.

```
HttpClient cl = HttpClient.newBuilder().build();

HttpRequest req = HttpRequest.newBuilder().
                       uri("https://character.com/isSpace").build();

HttpResponse res = client.send(request, myBodyHandler);
```

```
POST /isSpace HTTP 1.1
   Host: character.com

{ … "character":"@", …}
```

```
HTTP/1.1 200 OK
Deprecation: Tue, 31 Dec 2024

{ … "result":false, … }
```

OPENAPI INITIATIVE

```
deprecated: true
```

Approaches

Transfer

re-implement what works for static APIs

Reuse

use PL-mechanisms for remote APIs

Observation

inspect messages at runtime

Approach: Transfer

Java → Java

1. Identify method call
2. Check called method for deprecation
3. Report findings

Java → remote API

15+ API Clients in Java

How? OpenAPI-Spec?

Where? IDE? What if the deprecation is added later?

Approach: Reuse

- Generate a Java library for the HTTP endpoint automatically.
- Call `Character.isSpace` although it is a remote API.
- Make `isSpace` deprecated (Java) if the corresponding remote element is deprecated.

→ Easy, but only works if generated API clients are used/possible.
What if the deprecation is added later?

Approach: Observation

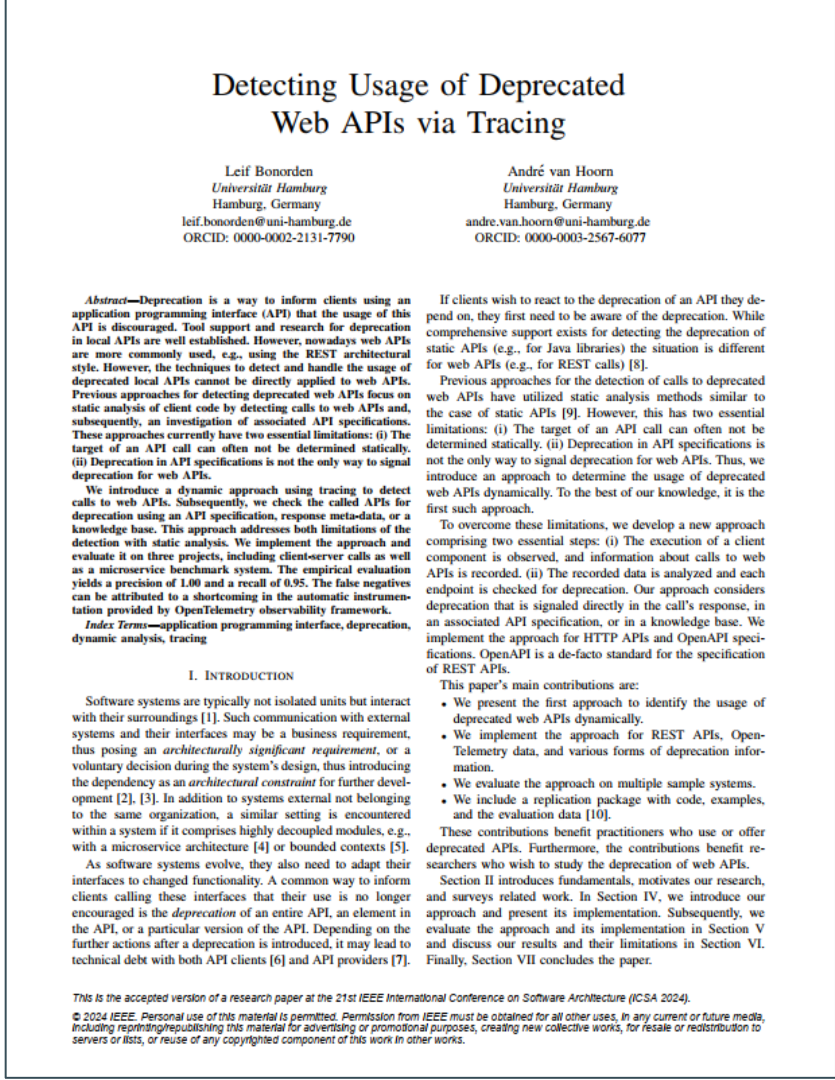
At runtime:

1. React on outgoing API calls.
2. Check dynamically: Is the endpoint deprecated?
3. Report findings!

Detecting Usage of Deprecated Web APIs via Tracing

Bonorden & van Hoorn, ICSA 2024

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG



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:

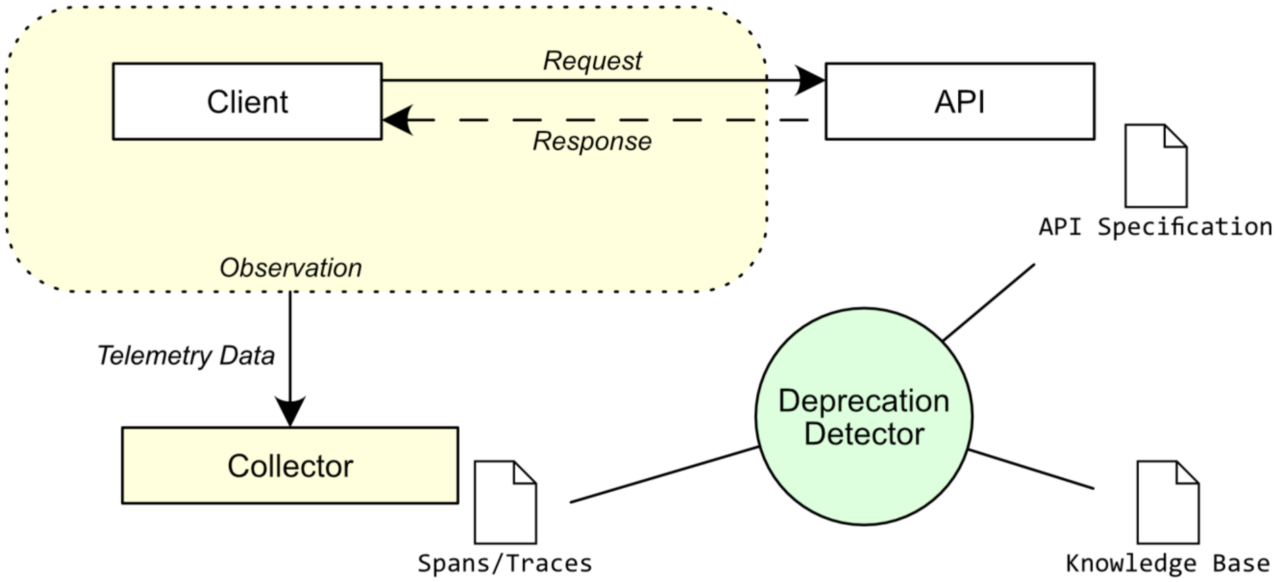
- We present the first approach to identify the usage of deprecated web APIs dynamically.
- We implement the approach for REST APIs, OpenTelemetry data, and various forms of deprecation information.
- We evaluate the approach on multiple sample systems.
- We include a replication package with code, examples,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

Our Approach with OpenTelemetry



Evaluation on Sample Projects

⊕

It works! ☺

It works well! ☺

- Precision 1.0
- Recall 0.95

⊖

- relies on instrumentation provided by OpenTelemetry
- only applicable dynamically (with typical disadvantages)
- **only evaluated on simple sample projects**

Outlook

this approach	▪ evaluation in industrial settings ▪ usage without telemetry data
general problem	▪ combination with static approaches ▪ improvements for API providers

Discussion

The API element
isSpace is deprecated.

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).