Universität Hamburg

Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics

Leif Bonorden

Understanding and Detecting Usage of Deprecated Remote APIs

An der Universität Hamburg eingereichte Dissertation

Hamburg, 2026

Advisors	Prof. Dr.-Ing. Matthias Riebisch[†] Universität Hamburg
	Prof. Dr.-Ing. André van Hoorn[†] Universität Hamburg
	Prof. Dr. Janick Edinger Universität Hamburg
	Prof. Dr. Justus Bogner Vrije Universiteit Amsterdam
Reviewers	Prof. Dr. Janick Edinger Universität Hamburg
	Prof. Dr. Justus Bogner Vrije Universiteit Amsterdam
	Prof. Dr.-Ing. Anne Koziolek Karlsruher Institut für Technologie
Additional members of the examination commission	Prof. Dr.-Ing. Mathias Fischer Universität Hamburg
	Prof. Dr. Fabian Kern Universität Hamburg
Date of the oral defense	21 July 2026

Leif Bonorden
Understanding and Detecting Usage of Deprecated Remote APIs
Dissertation, 2026

Universität Hamburg
Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics
Bundesstraße 56 B, 20146 Hamburg, Germany

Abstract

CONTEXT. Modern software systems reuse features from other software modules or services via Application Programming Interfaces (APIs). While deprecation mechanisms and automated tool support are well-established for local APIs (e. g., Java libraries), a significant gap exists for remote APIs (e. g., HTTP communication). Currently, there is no tool support to alert client developers when a remote API they call becomes deprecated.

OBJECTIVE. The primary objective of this dissertation is to improve the detection of deprecated remote API usage by designing and implementing a language-independent framework. This framework integrates multiple data sources to enable client developers to react effectively to evolving software ecosystems.

METHODS. Following a design science research approach, the dissertation includes a systematic mapping study of research on API deprecation to identify research gaps, an empirical quantitative and qualitative analysis of open-source Java repositories to characterize the usage of HTTP clients, an analysis of OpenAPI descriptions to evaluate how deprecation is documented in practice, and the construction, implementation and evaluation of a hybrid detection framework combining static and dynamic approaches.

RESULTS. The systematic study confirms that research on remote API deprecation is missing. Furthermore, the empirical analyses of HTTP clients in Java and deprecation practices in OpenAPI descriptions shows that any relevant solution needs to cover a variety of implementation options for client code and deprecation documentation. The hybrid detection framework achieves this widespread applicability by using auto-instrumentation for clients and combining multiple sources of deprecation documentation. Finally, the implemented prototype achieves a precision of 1.00 and recall values between 0.93 and 1.00 in controlled evaluations.

CONCLUSIONS. Automated, multi-source detection for usage of deprecated remote APIs is technically feasible and accurate. By combining dynamic runtime observation with static documentation analysis, the proposed hybrid approach provides a foundation for practical tools in evolving distributed systems.

Zusammenfassung

KONTEXT. Moderne Softwaresysteme nutzen Funktionen anderer Softwaremodule oder Dienste über Application Programming Interfaces (APIs). Während Deprecation-Mechanismen und automatisierte Werkzeugunterstützung für lokale APIs (z. B. Java-Bibliotheken) fest etabliert sind, besteht eine signifikante Lücke bei remote APIs (z. B. HTTP-Kommunikation). Derzeit gibt es keine Werkzeugunterstützung, um Client-Entwickler:innen zu warnen, wenn eine von ihnen aufgerufene remote API als deprecated markiert wird.

ZIEL. Das Hauptziel dieser Dissertation ist die Verbesserung der Erkennung der Nutzung von deprecated remote APIs durch ein sprachunabhängiges Framework. Dieses Framework integriert mehrere Datenquellen, um es Client-Entwickler:innen zu ermöglichen, effektiv auf sich entwickelnde Software-Ökosysteme zu reagieren.

METHODIK. Die Dissertation folgt einem Design-Science-Ansatz und umfasst eine systematische Klassifikation bisheriger Forschung über API-Deprecation zur Identifizierung von Forschungslücken, eine empirische quantitative und qualitative Analyse von Open-Source-Java-Projekten zur Charakterisierung der Nutzung von HTTP-Clients, eine Untersuchung von OpenAPI-Beschreibungen zur Evaluierung, wie Deprecation in der Praxis dokumentiert wird, sowie Entwurf, Implementierung und Evaluierung eines hybriden Erkennungs-Frameworks, das statische und dynamische Ansätze kombiniert.

ERGEBNISSE. Die systematische Studie bestätigt, dass Forschung zur Deprecation von remote APIs fehlt. Darüber hinaus zeigen die empirischen Analysen von HTTP-Clients in Java und Deprecation in OpenAPI-Beschreibungen, dass eine relevante Lösung eine Vielzahl von Implementierungsoptionen für Client-Code und Deprecation-Dokumentation abdecken muss. Das hybride Erkennungs-Framework erreicht diese breite Anwendbarkeit durch den Einsatz von automatischer Instrumentierung für Clients und die Kombination mehrerer Quellen für Deprecation-Doku-mentationen. Abschließend erzielt der implementierte Prototyp in kontrollierten Evaluierungen eine Precision von 1,00 und Recall-Werte zwischen 0,93 und 1,00.

SCHLUSSFOLGERUNGEN. Die automatisierte, quellenübergreifende Erkennung der Nutzung von deprecated remote APIs ist technisch umsetzbar und effektiv. Durch die Kombination dynamischer Laufzeitbeobachtung mit statischer Dokumentationsanalyse bietet der vorgeschlagene hybride Ansatz eine Grundlage für praktische Werkzeuge in sich entwickelnden verteilten Systemen.

Acknowledgements

I would like to express my sincere gratitude to all those who supported me during my doctoral studies.

I want to thank Matthias Riebisch for starting this journey with me, supporting and trusting me in finding a suitable topic, and enabling initial results. I want to thank André van Hoorn for taking over as an advisor, continued support, and enabling the central results of my dissertation. I want to thank Janick Edinger and Justus Bogner for jumping in without hesitation as soon as another solution was necessary, and for supporting me in finishing this work.

I would also like to thank the colleagues at the SWK research group for discussions, support and just keeping up some “normal” in disrupting times: Sebastian Gerdes, Alireza Hakamian, Oliver Kopp, Paula Rachow, Sandra Schröder, Stephanie Schulte Hemming, Tom Weber, Sebastian Werner, and Marion Wiese.

Most importantly, I thank my family for supporting my doctoral journey from enduring stressful phases to celebrating relieving accomplishments.

Contents

Part I

Foundations

1 Introduction

1.1 Motivation

Software has evolved from isolated programs to interconnected systems. This connectivity manifests itself on two levels. Internally, many systems consist of multiple components that work together as one logical unit. Externally, software systems often communicate with other systems via networks, for example, to exchange data or retrieve information in real time.

As systems become more interconnected, the points at which they interact become increasingly important. *Interfaces* define the boundary between two components or systems and specify how they communicate. In this way, they not only enable interaction, but also structure it and make it dependable over time.

At the same time, interfaces themselves are subject to change. As systems evolve, their interfaces may be extended, modified, or replaced. To allow timely adaptation to such changes, providers must communicate impending changes to their clients. A technical form of such communication is *deprecation*: By marking an element as *deprecated*, its provider signals that it should no longer be used, for example, because it will be removed.

Deprecation of Local Interfaces The following example imagines a software program that wants to display the local time in Hamburg (Germany). For this, it internally uses a software library[1] as shown in Listing 1.1.

Java

```
1 LocationId place = new LocationId("germany/hamburg");
2 TimeService service = new TimeService();
3 LocalDateTime result = service.getCurrentTimeForPlace(place);
```

Listing 1.1.: A Call to a Library Function in Java.

The client code instantiates objects for classes provided by the library (lines 1–2). The client then calls a method to obtain a desired value (line 3). How the library actually obtains, stores, or calculates the value is hidden behind its interface.

If the provider of this software library now encounters problems with their interface they may decide to change it. If a change leads to a situation in which clients should not use the interface any longer, this is signaled through deprecation on the providing component as shown in Listing 1.2.

[1] modeled after the timeanddate.com API, version 3, visited 2026-05-10

Java

```java
/**
 * Obtains the current local date and time for a specific location.
 *
 * @param location the unique identifier of the location
 * @return the current local date and time for the specified location
 *
 * @deprecated Use {@link #getCurrentTimeForLocation(LocationId)} instead.
 */
@Deprecated(since = "2.4.0", forRemoval = true)
public LocalDateTime getCurrentTimeForPlace(LocationId location) {
    return getCurrentTimeForLocation(location);
}
```

Listing 1.2.: Definition of a Deprecated Method in Java.

The library developer has indicated the deprecation in two places. In line 9, the deprecation itself is defined technically with the additional information that the version 2.4.0 is the first in which the method is deprecated and that this deprecation warns about a future removal of the method. In addition, line 7 uses another deprecation feature to also include information about deprecation in textual descriptions intended for client developers.

For the client developer, this deprecation now shows up in their development environment (IDE) if they still use the deprecated element as shown in Listing 1.3.

Java

```java
LocationId place = new LocationId("germany/hamburg");
TimeService service = new TimeService();
LocalDateTime result = service.getCurrentTimeForPlace(place);
```

Listing 1.3.: A Call to a Deprecated Method as Shown in an IDE.

The Java IDE checks each call for deprecation and signals usage of a deprecated method to the client developer by underlining and striking out the method name.

This alerts the client developer and prompts them to adapt their code as seen in Listing 1.4.

Java

```
LocationId location = new LocationId("germany/hamburg");
TimeService service = new TimeService();
LocalDateTime result = service.getCurrentTimeForLocation(location);
```

Listing 1.4.: The Updated Client Code in Reaction to the Deprecation.

The client has now removed the call to the deprecated method and has replaced it with the alternative method suggested in the textual documentation.

Deprecation of Remote Interfaces This practice has been well established for local interfaces, i. e., interfaces between the multiple components within a unit of execution. For the other type of interface, the one connecting a software system to other systems, a different situation arises: Since the two communication partners are not in the same local environment, they need to utilize network communication. To obtain the latest available data, they are not connected in advanced like the local interfaces, but actually establish the connection over the network at the time of execution. Now the calling client code looks different as presented in Listing 1.5.

Java

```
URI uri =
    URI.create("https://time.example/timeservice?placeid=germany/hamburg");
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder(uri).build();
HttpResponse<String> response =
    client.send(request, BodyHandlers.ofString());
LocalDateTime result = LocalDateTime.parse(response.body());
```

Listing 1.5.: A Call from Java Code to a Remote Interface.

This call uses an HTTP library to send a message to the URI specified in line 2. The response may take the form of `2026-12-31T23:59:59` and is then parsed to a Java object representing date and time.

This additional effort translates the call into an intermediate message format that is independent from the current local execution environment, but can be picked up by all sorts of systems. Such an intermediate format hides the actual interface of the called target. Instead, the provider of a service might issue an accompanying document describing how the interface should be used. Such a descriptive document is shown in Listing 1.6.

OpenAPI (YAML)

```yaml
openapi: 3.0.3
info:
  title: Time Service API
  version: 2.4.0
server:
  -  https://time.example
paths:
  /timeservice:
    get:
      summary: Get current time for a place
      parameters:
        - name: placeId
          in: query
          required: true
          schema:
            type: string
      responses:
        '200':
          description: OK
          content:
            text/plain:
              schema:
                type: string
```

Listing 1.6.: An OpenAPI Description of a Remote Interface.

This excerpt shows general information for the interface (lines 1–6), and information specific to the exact request (lines 7–16) and the expected response (lines 17–23).

While it is possible to indicate deprecation in this document, there does not exist a framework to enable an automated message to the client developer about this deprecation.

1.2 Objective

Problem Statement Early signaling of deprecation gives time for client developers to react in time before a remote API (application programming interface) is de-provisioned. However, without an automated approach to deliver such messages, it might never reach its intended recipients.

Thus, this dissertation is concerned with the introduction of automated deprecation detection for remote APIs – similar to the mechanisms for local APIs:

Improve	the detection of deprecated remote API usage
by	designing and implementing a framework
that	provides language-independent deprecation analysis of dynamic calls and integrates multiple data sources
in order to	enable client developers to effectively react to evolving ecosystems.

Chapter 4 will discuss the details for this problem statement and will introduce accompanying research questions.

Scope The dissertation focuses the problem on the specific situation of Java code calling remote HTTP APIs. Within this setting, the goal is to construct a deprecation detection framework as general as possible.

As such a deprecation framework is a tool for human developers and other stakeholders, the technical quality is only the basis for actual usability. However, as the first approach to this problem, this dissertation concentrates on the technical feasibility – while, nevertheless, keeping the human perspective in mind.

Furthermore, the dissertation focuses on deterministic and reliable solutions. Thus, approaches including generative artificial intelligence or machine learning are only discussed briefly.

The goal for the contribution in this written document is a concise presentation. Thus, only contributions directly related to the problem statement and objective are included. Other related results from earlier publications are not repeated here.

Research Approach The main tool for this project is design-based research. This research method works in iterations to solve relevant problems with the background of scientific knowledge. To guide the design process for the deprecation detection framework, multiple preceding studies are conducted to foster the understanding of the problem in its entirety. Chapter 4 presents the methodical approach in more detail.

From a philosophical point of view, the research changes between a positivistic and a pragmatic mode. While the core technical problem and solution pose an objective task with its own quality criteria and can be seen as positivistic science, the entire research process aims to use this positivistic element in a greater environment to evaluate its usefulness in practical scenarios. This is reflected in the main parts of the thesis where a

positivistic stance prevails until the the result is seen in a greater context to open up the discussion to the pragmatic perspective.

1.3 Structure

The dissertation comprises four parts: While Part I (Foundations) introduces the fundamentals for the topic, Part II (Static Deprecation Analysis) and Part III (Dynamic Deprecation Analysis) evaluate design options in parallel to each other, and Part IV (Results and Discussion) constructs the solution based on the preceding parts.

Part I – Foundations The first part introduces the topic in Chapter 1 (Introduction), and presents the technical background in Chapter 2 (Background). In addition, the related work on the core topic is studied systematically in Chapter 3 (Systematic Analysis of Scientific Work on API Deprecation) to derive the detailed goal and research questions in Chapter 4 (Research Goal, Approach and Contribution).

Part II – Static Deprecation Analysis The second part analyzes the practices of HTTP calls in Java as well as their detection in Chapter 5 (Static Analysis of HTTP Clients in Java). Complementary, the situation of API providers and their documentation of deprecation is studied in Chapter 6 (Static Analysis of Remote APIs). Conclusively, the two perspectives are combined to evaluate the feasibility of static approaches to the overarching problem in Chapter 7 (Discussion of Static Approaches).

Part III – Dynamic Deprecation Analysis The third part uses the same structure as the second part, but with dynamic analyses. First, clients and their requests are observed in Chapter 8 (Dynamic Analysis of HTTP Clients in Java) and, subsequently, providers and their responses are investigated in Chapter 9 (Dynamic Analysis of Remote APIs). In conclusion, the two perspectives are joined in Chapter 10 (Discussion of Dynamic Approaches).

Part IV – Results and Discussion The fourth part moves from understanding the problem to constructing the solution. Therefore, the deprecation detection framework is presented in Chapter 11 (Hybrid Approach). Thereafter, the framework and its implementation are evaluated in Chapter 12 (Evaluation) and further discussed in Chapter 13 (Discussion). Finally, a summary as well as future work are presented in Chapter 14 (Conclusions).

A cross-cutting concern is given by the distinction between clients with their requests on the one hand and providers with their responses on the other hand: Chapters 5 and 8 (the first chapters of Part II and Part III, respectively) are concerned with the analysis of clients. Chapters 6 and 9 study the analysis of providers.

Background

2

This chapter offers a concise introduction to the technical concepts for the dissertation. The goal is to put them into perspective and distinguish them from each other. The subsequent Chapter 3 complements the introduction by studying the related work on API deprecation systematically from a meta-perspective.

2.1 Application Programming Interfaces (APIs)

2.1.1 Software Architecture

The topic of interest for this dissertation lies in the discipline of Software Architecture. The scope of this discipline can be defined from two different perspectives. The first one offers a structural view:

> SOFTWARE ARCHITECTURE (BASS ET AL.)
>
> The *software architecture* of a system is the set of structures needed to reason about the system. These structures comprise software elements, relations among them, and properties of both. (Bass et al., 2022)

In contrast, the second definition is based on the process to achieve the structural outcome:

> SOFTWARE ARCHITECTURE (TAYLOR ET AL.)
>
> A software system's *architecture* is the set of principal design decisions made about the system. (Taylor et al., 2009)

Both lead to the essential observation, that software architecture is a concept that is individual to each software system. General patterns may emerge, but in its core, software architecture is system design based on quality criteria:

> FIRST LAW OF SOFTWARE ARCHITECTURE
>
> Everything in software architecture is a trade-off. (Richards and Ford, 2020)

Components and Connectors One way to build a software architectural structure based on individual decisions is given by a model of components and connectors:

COMPONENT

A *software component* is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context. (Taylor et al., 2009)

CONNECTOR

A *software connector* is an architectural element tasked with effecting and regulating interactions among components. (Taylor et al., 2009)

After defining components and connectors for a software system in isolation, the configuration connects the concepts to achieve the overall software architecture:

CONFIGURATION

A *architectural configuration* is a set of specific associations between the components and connectors of a software systems's architecture. (Taylor et al., 2009)

Since this dissertation is concerned with communication between software components, we will focus the connectors in this setting.

Software Quality To guide the design of all elements in a software architecture, the desired quality criteria need to be elicited as requirements. For such quality criteria, multiple models exist, e. g., (ISO/IEC 25010, 2023). From this standard, we will mostly need the following quality criteria to design our deprecation detection framework or to reason about the systems we apply the framework to:

FUNCTIONAL SUITABILITY

capability of a product to provide functions that meet stated and implied needs of intended users when it is used under specified conditions (ISO/IEC 25010, 2023)

PERFORMANCE EFFICIENCY

capability of a product to perform its functions within specified time and throughput parameters and be efficient in the use of resources under specified conditions (ISO/IEC 25010, 2023)

COMPATIBILITY

capability of a product to exchange information with other products, and/or to perform its required functions while sharing the same common environment and resources (ISO/IEC 25010, 2023)

MAINTAINABILITY

capability of a product to be modified by the intended maintainers with effectiveness and efficiency (ISO/IEC 25010, 2023)

2.1.2 Connectors

One design option for connectors are interfaces:

INTERFACE

shared boundary between two functional units, defined by various characteristics pertaining to the functions, physical interconnections, signal exchanges, and other characteristics, as appropriate (ISO/IEC 2382, 2015)

Interfaces are special connectors as they are not implemented as standalone elements of an architecture, but are coupled to a component.

We discuss three main concepts for connectors and interfaces (Hohpe and Woolf, 2013; Taylor et al., 2009):

Procedure Calls Procedure calls are a basic concept of interaction in programming languages. Depending on the language they are also called method invocations.

Remote Procedure Calls mimic local calls within a programming language, but actually invoke a program on a different machine. This concept covers special instantiations for specific programming languages (e. g., Java Remote Method Invocation) and communication specification (e. g., SOAP).

Shared Data Access Another form of interaction between components is the use of common resources such as shared databases, or explicit exchange files. Such cooperation requires close communication to prevent unintended modification of data.

Messaging Systems In distinction to the procedures mentioned above, messaging system provide a middle-ware technology to transmit events in messages between components. While procedure calls usually wait for an immediate response, events are typically sent without such expectation.

Application Programming Interfaces Application programming interfaces can be seen as a kind of procedure call using an intermediate language. The original use for the term was published in the context of communication abstraction (allowing the change of implementation while keeping the interface) in the context of interaction of an application program with the computer hardware through system software: “hardware independence at the central computer means that a consistent *application program interface* could be maintained if that computer were replaced” (Cotton and Greatorex, 1968). A modern interpretation is given by the following definition:

> Application Programming Interface (API)
>
> We define an *API* to be the interface to a reusable software entity used by multiple clients outside the developing organization, and that can be distributed separately from environment code. (Robillard et al., 2013)

Often, small deviations from this definitions are accepted. In such a broader sense, the concept of APIs is used to also describe interfaces that do not have multiple clients outside the developing organization (yet).

The interaction through an API includes two parties and two messages between them: An API client sends a request and the API provider returns a response.

Instead of “clients and providers” several other terms are used, e. g., “clients and developers of an API” (Robillard et al., 2013), “consumers and providers” (Lercher et al., 2024), “users and developers of an API” (Lamothe et al., 2022), “developers and API suppliers” (Meng et al., 2018).

APIs can essentially be split into two categories: local APIs within the same environment, and remote APIs that require the use of network infrastructure.

2.1.3 Local APIs

The concept of ”local APIs" is often used without a precise definition, but as the complementary type to remote APIs (Zimmermann et al., 2023). The same concept is also known as “static APIs”, because they are statically resolved (Raatikainen et al., 2021), or “Programming Language APIs”, because APIs in programming languages are the most important subtype for this category (Liu et al., 2020), or “library APIs”, because they are used to access additional libraries in programming environments (Lamothe et al., 2022), or “traditional APIs” to contrast the newer remote APIs (Wittern et al., 2017).

However, local APIs are not limited to individual programming languages, but can also exist in ecosystems supporting multiple programming languages. Important examples include the Android mobile operating system (e. g., with programming languages Java and Kotlin) or the Java JVM execution environment (e. g., with programming languages Java, Clojure and more).

Another concept that shares the same name "local APIs" are local versions of remote APIs (e. g., if they are not yet published APIs), (de Souza et al., 2004), or APIs used only by a small client group (Lima and Hora, 2020). Within this dissertation we do not consider these interpretations of the term.

Documentation of Local APIs Local APIs are usually documented within the source code that defines them. In Listing 1.2, an example for Javadoc was shown. Tools may extract such information from the source files to generate an external documentation document for developers.

2.1.4 Remote APIs

In contrast to local APIs, remote APIs appear in the context of distributed systems:

> DISTRIBUTED SYSTEM
>
> A *distributed system* is a software architecture that consists of multiple interconnected nodes or servers working together to achieve a common goal. These nodes communicate with each other over a network and coordinate their actions to provide a unified and scalable computing environment. (Unmesh, 2024)

Using this setting, we can define remote APIs:

> REMOTE API
>
> A *remote API* is a set of well-documented network endpoints that allow internal and external application components to provide services to each other. These services help achieve domain-specific goals, for instance, fully or partially automating business processes. They allow clients to activate provider-side processing logic or support data exchanges and event notifications. (Zimmermann et al., 2023)

As an alternative term, remote APIs are also known as "WebAPIs", because they are also the basis for communication through the world wide web (Raatikainen et al., 2021).

Types of Remote APIs We briefly distinguish some standard communication formats and styles for remote APIs.

The most common communication protocol is the Hypertext Transfer Protocol (HTTP), and the most common style in which is it used is Representational State Transfer (REST). We will introduce this type in the subsequent chapters and provide examples throughout dissertation.

Two other forms of modern remote APIs are given by GraphQL and gRPC. They use a similar underlying structure as HTTP/REST, but structure their messages and communication differently. We will discuss the generalization of our deprecation detection framework to these style in Section 13.1.

Documentation of Remote APIs In contrast to local APIs, remote APIs cannot be documented for their clients in their source code since that source code is usally not visible to the clients. Instead, external description languages are used, e. g., an interface description language (IDL) for CORBA or OpenAPI to describe HTTP/REST APIs. Again, we will introduce OpenAPI in more detail in the relevant chapters with examples throughout the thesis.

2.2 API Evolution & Deprecation

2.2.1 API Evolution

Similar to software system and components, an API might change over time to account for changes in the element it accesses, to correct technical mistakes, or to improve the overall design of the system. Thus, an API might change with or independent to its implementation.

Some changes of an API require a reaction from their clients because the former interaction is no longer possible. Such changes are called “breaking changes” because they break the clients functionality.

To warn about breaking changes or other unexpected behavior of an API, the API documentation may use *deprecation* to recommend *not* using the respective API element.

2.2.2 API Deprecation

The exact meaning of deprecation varies between contexts. The following characterization from the Java programming language offers a general view:

> DEPRECATION
>
> Programmers are sometimes discouraged from using certain program elements (modules, classes, interfaces, fields, methods, and constructors) because they are considered dangerous or because a better alternative exists. (Gosling et al., 2025)

This general concept can be split into two interpretations, that may occur after each other: In the first option, deprecation is issued as a general warning that using the deprecated element might not have the intended behavior (e. g., a local API might be not thread-safe). The second option, sometimes called "sunsetting" (Wilde, 2019b) announces that the API element will be removed from the API in a future version.

Deprecation as a Process The general API deprecation process involves providers and clients of an API and comprises four steps (Bonorden and Riebisch, 2022a):

- **Decision.** A provider decides to deprecate.
- **Documentation.** A provider documents deprecation.
- **Information.** A client finds out about deprecation.
- **Reaction.** A client reacts to deprecation.

In particular, this four-step model combines the perspectives of providers and clients. As APIs are means for interaction between (sub-)systems, they are also relevant for the interaction between providers and clients.

In addition to the deprecation itself, its documentation can convey additional information e. g., if sunsetting is planned, since when the API element is deprecated, when the element will be removed from the API, an alternative to use, or a reason for the deprecation (Sawant et al., 2018a).

Identifying Calls to Deprecated APIs Detecting usage of deprecated APIs comprises three essential steps (Bonorden and van Hoorn, 2024a):

1. Detection of an API call and the respective target for that call
2. Analysis of the deprecation status for the call target
3. Reporting of findings

For local APIs, in the example of the Java programming language, these three steps are executed by the compiler (supported by a dependency manager if necessary) as it has access to all relevant information within the local ecosystem. An IDE may pick up the compiler's output and show the deprecation to the client developer as seen in Listing 1.3.

For remote APIs, the same three essential steps can be conducted. However there is no continuous tool support that would enable a complete automatic process. Instead, a manual analysis would be required. Thus, this dissertation aims to address this gap.

Finally, deprecation of remote APIs poses an essential difference to deprecation of local APIs: For local APIs, a compiler or another build tool merges the client code and the functionality provided by the API into a single unit that can later be executed. This ensures constant executability independent of any change in an API's deprecation status. In contrast, remote APIs are only contacted at runtime. Thus, also a deprecation that is introduced after the client's development has concluded is similarly important.

2.3 Related Topics

We briefly survey work that is not directly concerned with API deprecation, but is related to the concept. The subsequent Chapter 3 will systematically study related work concerning API deprecation directly and in more detail.

API Design As good API design can reduce the need for breaking changes (Tulach, 2012), it is related to the topic of deprecation. However, even with good API design, changes to an API can never be ruled out completely (Murphy et al., 2018).

Such API design literature covers general approaches like "API first" (Spichale, 2025) or presents design patterns for APIs Zimmermann et al., 2023. General software design literature, e. g., Domain-Driven Design (Evans, 2003; Vernon, 2013; Vernon, 2016), also provides processes that influence the API design. Another research direction is given by checking the adherence to underlying principles, e. g., for REST APIs (Bogner et al., 2024a).

Since API design is not central to this dissertation's topic, we will only discuss it in the context of the need for deprecation as part of the systematic study of related work on API deprecation in Chapter 3.

Understanding and Using APIs In addition to its design, the APIs documentation is central for client developers to understand the API. In particular, documentation needs to feature information on deprecation. Good documentation ensures clients use API as indicated, making it easier for providers to reason about usage and change impact. Research on API documentation studies what client developers need from documentation, e. g., (Meng et al., 2018), what is actually documented (Maalej and Robillard,

2013), and what influences the understandability of an API (Bogner et al., 2023; Bogner et al., 2024b).

If the actual use does not match the intended use, the API is misused. Such misuse of API makes it hard to follow suggested changes as a reaction to deprecation. Furthermore, if bad usability is one reason to change APIs later, potentially triggering deprecation. However, this dissertation focuses the case in which deprecation is actually present – independent of the causes.

API Maintenance and API Evolution As deprecation usually occurs in the context of evolution, also topics connected to API evolution in general are relevant to API deprecation. Research topics mostly cover patterns of evolution (Lübke et al., 2019), adapting clients to API changes, migrating from one API to another, and API reuseLamothe et al., 2022.

Directly connected to deprecation is the topic of API versioning. Popular versioning formats like semantic versioning include rules how removal of an API element requires prior deprecation. (Serbout and Pautasso, 2024b).

These topics directly relevant to API deprecation will be covered by the systematic analysis in the subsequent chapter Chapter 3.

Further API Qualities Similar to general software systems, APIs may feature a variety of quality criteria. In particular, API performance (El Malki and Zdun, 2023), API reliability (El Malki et al., 2022) and API Security (Madden, 2021) are studied. Reengineering and optimization of a system or its API may improve these quality criteria, but may also require changes in the API itself and thus may trigger deprecation

These qualities are not relevant to this dissertation and thus, are not further discussed.

3 Systematic Analysis of Scientific Work on API Deprecation

Following the discussion of the technical background and related topics, this chapter identifies and presents related work on API *deprecation*. While this systematic approach yields one of the thesis's contributions, it precedes the main parts of the thesis to structure the research field in advance and to identify research gaps that will motivate the thesis's goal in Chapter 4.

Publication Note This chapter comprises the study *"API Deprecation: A Systematic Mapping Study"* published in (Bonorden and Riebisch, 2022a). The artifact (Bonorden and Riebisch, 2022b) presents supplementary material. Furthermore, the extended abstract (Bonorden and Riebisch, 2023) also covers these results.

3.1 Research Design

3.1.1 Research Objective

To survey the current state of research and to identify gaps within, the following research questions guide the empirical meta-study:

RQ 0 **What is the state of research on the deprecation of local and remote APIs?**

RQ 0.1 Which addressed beneficiaries, applied research strategies, and types of contribution do publications on API deprecation present?

RQ 0.2 What types of APIs and which aspects of deprecation do publications on API deprecation address?

RQ 0.3 Which gaps remain in research of API deprecation?

While research question *RQ 0.1* follows the *Who-What-How framework* (Storey et al., 2020) to provide a general meta perspective (i. e., not specific to API deprecation), research question *RQ 0.2* adds two dimensions specific to API deprecation. Together they allow the construction of a systematic map (Petersen et al., 2008) specific to API deprecation. Finally, research question *RQ 0.3* derives research gaps from the systematic map. Collectively, the three research sub-questions answer the broader research question *RQ 0*.

Research question *RQ 0* will contribute to the definition of the core research questions *RQs 1–4*. All research questions will be presented and discussed in Section 4.1.

3.1.2 Research Method

A *systematic mapping study* corresponds to the research questions in this chapter: It is a form of meta-study that categorizes primary studies and visualizes such categorization as a map of the research-field.

Adopted from meta-methods of evidence-based medicine, the paradigm of *evidence-based software engineering* aims "to provide the means by which current best evidence from research can be integrated with practical experience and human values in the decision making process regarding the development and maintenance of software" (Kitchenham et al., 2004). At its core, evidence-based software engineering utilizes published research results ("primary studies") to summarize insights, identify gaps, or build a systematization ("secondary studies") (Kitchenham, 2004). In contrast to *ad-hoc reviews*, evidence-based software engineering requires a systematic process for the identification of relevant primary studies to ensure coverage of the research field (Ralph and Baltes, 2022).

Kitchenham and Charters (2007) describe three main stages common for all types of systematic meta-studies: planning, conducting, and reporting. While this Section 3.1 presents the planning stage, the subsequent Section 3.2 describes the conducting stage. For the reporting stage, Sections 3.3 and 3.4 present findings and discuss them, respectively.

As a special kind of systematic meta-study, systematic mapping studies take a rather broad point of view to categorize primary studies or to find gaps in the research field. These goals give rise to two "mapping" processes: mapping studies to categories and creating a map that might show gaps. Outside of software engineering research, systematic mapping studies are also known as scoping reviews (Pham et al., 2014) or evidence maps (Miake-Lye et al., 2016).

Adapting the method to software engineering, Petersen et al. (2008) formulated guidelines for conducting systematic mapping studies. Subsequently, Petersen et al. (2015) updated these guidelines: Following the definition of the research questions, their process comprises three essential steps: the search for primary studies in databases or publication indices, the selection of relevant studies based on pre-defined criteria, and the categorization leading to a map. In addition to a database search, snowballing uses a primary study's references and citations to find further candidates (Wohlin, 2014). The systematic mapping process for the present study is shown in Fig. 3.1.

Ralph and Baltes (2022) discuss that systematic mapping studies "typically fail to fulfill any of the core purposes of systematic reviews" (e. g., "analyzing, synthesizing or critiquing primary studies"). Furthermore, they argue that systematic mapping studies "are essentially works-in-progress" (e. g., as a "pilot study for a more ambitious systematic review"). On the contrary,

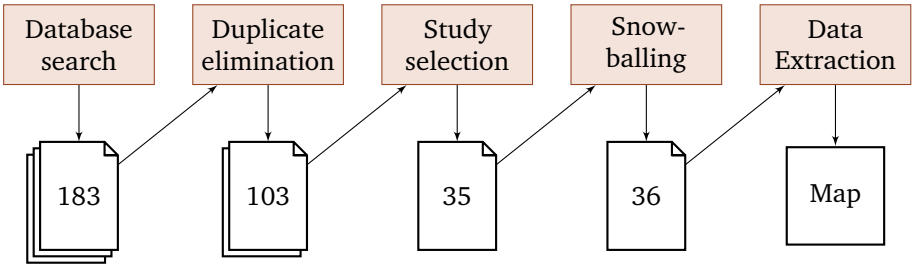

Fig. 3.1.: Systematic process of study identification and study mapping

Kitchenham et al. (2011) see potential for systematic mapping studies to be "valuable research outcomes in their own right". While this systematic mapping study has been published individually (Bonorden and Riebisch, 2022a), its purpose of gap identification meets the work-in-progress idea to develop goal and approach for this thesis in Chapter 4. Subsequently, Chapters 5 and 6 analyze the results from the systematic mapping study further as they review relevant primary studies on local APIs to discuss their transferability to remote APIs.

3.2 Mapping of Primary Studies

3.2.1 Selection of Primary Studies

Search Strategy The initial search for primary studies considers five targets (two publisher databases and three independent indices) and the search terms "API deprecation" and "deprecated API". Table 3.1 details the inquiries as they are adapted to the targets' specific search options.

This search yields 183 results. After the elimination of duplicates, a set of 103 primary studies remains.

Inclusion and Exclusion Criteria To filter the set of results further, inclusion and exclusion criteria govern the selection of primary studies. The following inclusion criteria describe the kind of primary studies the secondary study aims for:

- I1 **Publication type.** The study was peer-reviewed and published in a scientific journal, or in conference or workshop proceedings.
- I2 **Research subject.** The study discusses API deprecation.
- I3 **Language.** The study is written in English.

However, even if all inclusion criteria apply, matching one of the following exclusion criteria leads to dismissal of the primary study:

E1 **Missing contribution.** The study does not feature new insights (neither knowledge nor solutions) on API deprecation.

E2 **Extended.** Another primary study is an extended version of this publication.

Following the idea of a map of the entire research area, we choose permissive criteria. In particular, I1 includes all tracks at any conference or workshop, e. g., tool demonstrations. However, we have to add the restriction E1 as deprecated APIs are often mentioned in studies that do not contribute to the state of research on API deprecation, e. g., deprecation mentioned in an introduction or as a problem encountered during an implementation task. Additionally, the restriction E2 prevents the same results from appearing multiple times which would distort the systematic map (Kitchenham and Charters, 2007). We refrain from generic limitations, e. g., using a cut-off date. We apply the inclusion and exclusion criteria using a adaptive reading depth (Petersen et al., 2008): If title, abstract and keywords allow a confident decision, we decide based on these elements. Otherwise, we screen the full text to come to a decision.

This step in the process features two researchers, L. Bonorden and M. Riebisch, to mitigate researcher's bias. The researchers discuss any differences in their assessments and come to an agreement.

The application of inclusion and exclusion criteria leads to a set of 35 primary studies.

Snowballing Publications that are referenced in the selected primary studies are additional candidates that might be relevant to the systematic mapping study. Furthermore, this applies also to newer publications that cite one of the selected primary studies. Backward and forward snowballing (Wohlin, 2014) on the Semantic Scholar Academic Graph (Ammar et al., 2018) retrieves these additional 1033 candidates. Applying inclusion and exclusion criteria with adaptive reading depth eliminates all but 1 publication (Spoon, 2007).

Tab. 3.1.: Individual Inquiries per Target with Search String, Search Scope, and Number of Results

Database/Index	Search String	Search Scope	Number of Results
ACM Digital Library	"api deprecation"	Full text	22
ACM Digital Library	"deprecated api"	Full text	46
IEEE Xplore	api AND deprecation	All metadata	13
IEEE Xplore	api AND deprecated	All metadata	26
Microsoft Academic	"api deprecation"	Full text	18
Microsoft Academic	"deprecated api"	Full text	19
Science Direct	api AND deprecation	Title, abstract, keywords	4
Science Direct	api AND deprecated	Title, abstract, keywords	6
Web of Science	api AND deprecat*	Title, abstract, keywords	32

Following the inclusion of an additional study, we reconsider the search strategy and selection criteria as they have failed to identify this publication. However, the additional study has been published at a small workshop, does not feature a DOI or other citable identifier, and is not indexed in any major database or research index. Thus, we conclude that the addition of another study does not indicate a fault of the search strategy or selection criteria of our study.

The set of primary studies now contains 36 publications.

Verification of the Search Process In addition to snowballing, a validation set of pre-selected articles can verify the search process further. For this study, five articles (Brito et al., 2018; Haryono et al., 2021; Li et al., 2020; Sawant et al., 2018a; Yasmin et al., 2020) form the validation set that the search result should include. As the set of 36 primary studies actually contains all of these articles, this affirms the research process.

Selected Primary Studies The search process yields 36 primary studies that are classified in the following subsection. Section .0.0 features a complete list of these selected studies together with their individual classifications.

Fig. 3.2 shows the chronological distribution of primary studies. While the first publication is from 2005, 83% are issued since 2016 with a steep increase in 2018.

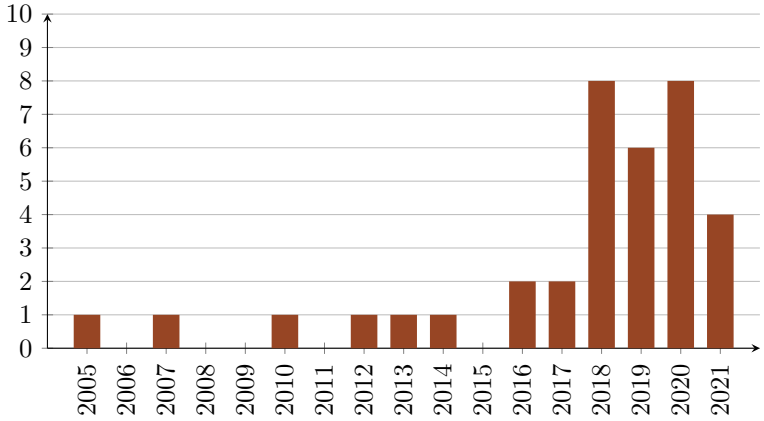

Fig. 3.2.: Years of Publication for the Selected Primary Studies

3.2.2 Classification of Primary Studies

Following research question RQ 0.1, the first three classifications are not API-specific. Research questions RQ 0.2 adds the remaining two classifications:

Beneficiaries – *Who?* Storey et al. (2020) distinguish software engineering research by the intended recipient:

- **Human stakeholders.** The results help people involved in software development to fulfill their tasks, e. g., provision of information for developers.
- **Technical systems.** The results improve the quality of software systems, e. g., enhanced performance for a tool.
- **Researchers.** The results support further academic or industrial studies, e. g., synthesis of the state of research.

A single study may address more than one intended recipient. A publication may state the beneficiary explicitly or suggest it in the motivation section, an example, or some other part. We follow (Storey et al., 2020) in coding *human stakeholders* and *technical systems* rather liberally, but we use *researchers* only if the intention is clear.

The selected primary studies address human stakeholders in 34 out of 36 studies (approximately 83 %), technical systems in 5 out of 36 studies (approximately 14 %), and researchers in 7 out of 36 (approximately 19 %). Fig. 3.3 illustrates these numbers.

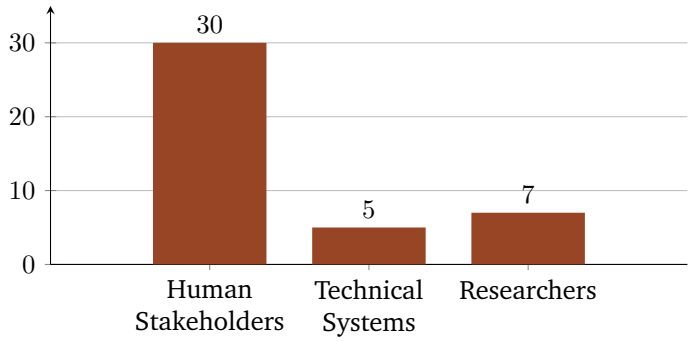

Fig. 3.3.: Intended Beneficiaries of the 36 Primary Studies

Contributions – *What?* Storey et al. (2020) further distinguish studies based on the mode of research (Stol and Fitzgerald, 2020), i. e., knowledge-seeking (i. e., descriptive, understanding problems and situations) or solution-seeking (i. e., constructive, addressing engineering problems). For this classification, selected primary studies show three sub-types of solution-seeking contributions:

- **Tool.** An implemented and executable artifact to perform actions automatically. *Example:* a program checking a Python library for deprecated APIs. (Vadlamani et al., 2021).
- **Method.** Instructions on how to solve a problem manually or conceptually. *Example:* a technique on inlining deprecated methods prior to refactoring (Perkins, 2005).
- **Classification.** A framework for categorization to assist decision making in practice. *Example:* a categorization for browser features to guide their potential deprecation efforts (Mirian et al., 2019).

A single study may provide more than one type of contribution. In particular, it can also contain more than one solution-seeking contribution. Publications explicitly state the study's contributions.

The selected primary studies include knowledge-seeking contributions in 21 out of 36 studies (approximately 58%) and solution-seeking contributions in 19 out of 36 studies (approximately 53%)[1]. In more detail, 14 out of 36 studies present tools (approximately 39%), 5 out of 36 studies present methods (approximately 14%), and 1 out of 36 studies presents a classification (approximately 3%). Fig. 3.4 illustrates these numbers.

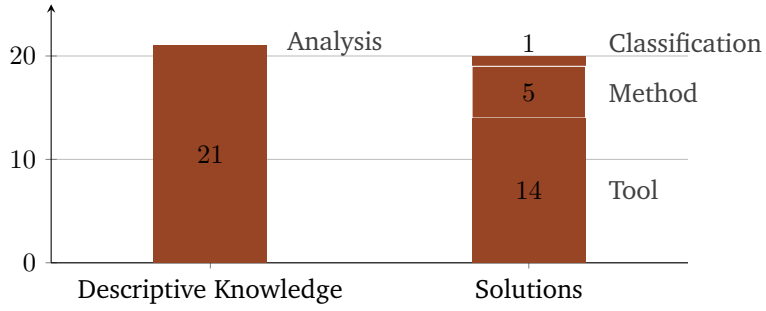

Fig. 3.4.: Types of Contribution (Descriptive or Constructive) in the Primary Studies

[1]The publication (Bonorden and Riebisch, 2022a) falsely gives this number as 56%.

Research Strategies – *How?* Storey et al. (2020) finally consider strategies of empirical and non-empirical research. Such strategies are a broader categorization than research methods:

- **Field.** An investigation in a natural setting, e. g., performing a field study.
- **Lab.** An observation of test subjects in a highly controlled setting, e. g., during a lab experiment.
- **Respondent.** A setting-independent collection of personal answers, e. g., through a sample survey.
- **Data.** An analysis without human actors, e. g., by implementing an automated tool.
- **Non-empirical.** An assessment without (direct) involvement of empirical data, e. g., proving a formal theorem.

A single study may follow more than one research strategy. Publications introduce the study's research method, and these methods relate to a broader research strategy.

The selected primary studies exercise empirical data strategies in 31 out of 36 studies (approximately 86%), empirical respondent strategies in 5 out of 36 studies (approximately 14%), empirical field strategies in 2 out of 36 studies (approximately 6%), and non-empirical strategies in 1 out of 36 studies (approximately 3%). No use of empirical lab strategies occurs. Fig. 3.5 illustrates these numbers.

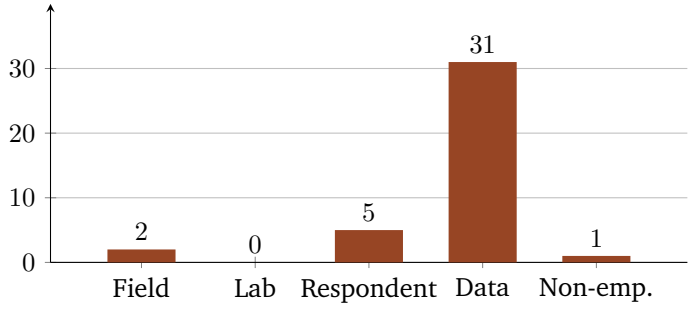

Fig. 3.5.: Research Strategies Identified in the 36 Primary Studies

API Types Local and remote APIs as introduced in Section 2.1 are the primary distinction for this classification. Both further fall into subclasses related to ecosystems, e. g., programming languages or technical platforms:

- **Java.** The programming language Java together with associated systems (e. g., the Maven build tool and Maven Central Repository).
- **JavaScript.** The programming language JavaScript with the npm package manager and registry.
- **Python.** The programming language Python with pip, uv, and other package managers.
- **HTTP/REST.** Web APIs using the REST architecture style for communication via the HTTP protocol.
- **Android.** The software development kit, platform and ecosystem for the mobile operating system Android.
- **other local APIs.** Ecosystems each appearing only once, i. e., C/C++, C#, Chrome, Pharo, Smalltalk.

A single study may address more than one type of API.

The selected primary studies address local APIs in 35 out of 36 studies (approximately 97%), while including remote APIs (HTTP/REST) only in 1 out of 36 studies. In more detail, the studies research addresses Java in 17 out of 36 studies (approximately 47%), Android in 9 out of 36 studies (approximately 25%), Python in 4 out of 36 studies (approximately 11%), JavaScript in 2 out of 36 studies (approximately 6%). Fig. 3.6 illustrates these numbers.

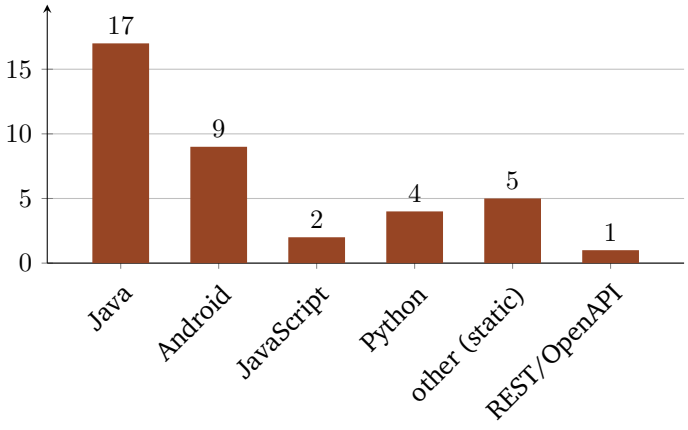

Fig. 3.6.: API Types Studied in the 36 Primary Studies

Aspects of Deprecation This category follows the four-step model of deprecation introduced in Section 2.2:

- **Decision.** A provider decides to deprecate.
- **Documentation.** A provider documents deprecation.
- **Information.** A client finds out about deprecation.
- **Reaction.** A client reacts to deprecation.

A single study may address more than one aspect of deprecation.

The selected primary studies investigate the provider's decisions to deprecate in 3 out of 36 studies (approximately 8%), the provider's documentation of deprecation in 9 out of 36 studies (25%), a client seeking information about deprecated APIs in 2 out of 36 studies (approximately 6%), and a client's reaction to deprecation in 27 out of 36 studies (75%). Fig. 3.7 illustrates these numbers.

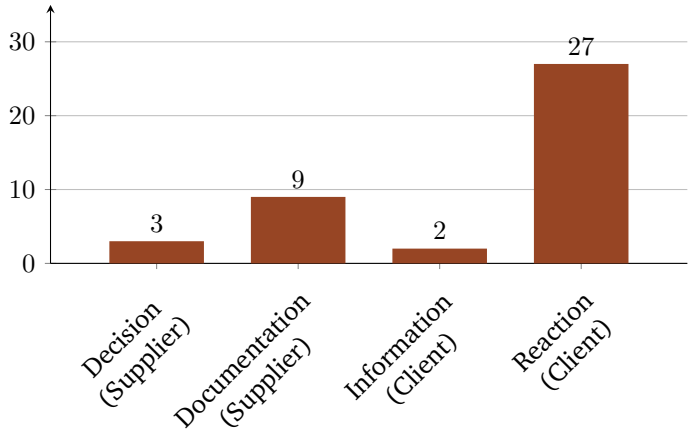

Fig. 3.7.: Aspects of Deprecation Discussed in the 36 Primary Studies

3.2.3 Map of the Research Field

A systematic map is a two-dimensional representation of classified primary studies. Hence, choosing the dimensions to model defines the map. To visualize such a map, bubble plots are the most used approach (Petersen et al., 2015).

For the research field of API deprecation, we analyze the results of the five classifications and decide to use both of the API-specific classifications (API types and aspects of deprecation) as well as the type of contributions ("What?") for the systematic map. We exclude the beneficiaries ("Who?") as the combination with any other dimension does not show additional insights. Furthermore, we exclude the research strategies ("How?") because data strategies prevail for each combination with another dimension.

Fig. 3.8 shows the resulting map. The map's left-hand side shows the 36 primary studies with respect to the dimensions *Type of Contribution* and *Aspect of Deprecation*. Furthermore, the map's right-hand side shows the same 36 primary studies with respect to the dimensions *API type* and *Aspect of Deprecation*.

Furthermore, we are now able to answer the research questions RQ 0.1 and RQ 0.2.

Answer to RQ 0.1 *Which addressed beneficiaries, applied research strategies, and types of contribution do publications on API deprecation present?*

Research on API deprecation mainly intents to benefit human stakeholders and their decisions more than direct improvements for technical systems. Studies are evenly split between descriptive and constructive contributions. Both types of contributions primarily employ data research strategies.

Answer to RQ 0.2 *What types of APIs and which aspects of deprecation do publications on API deprecation address?*

Research on API deprecation strongly focuses clients' reactions to deprecation in the Java programming language and Android ecosystem. On the client side of deprecation, other types of (local) APIs appear in less studies. Studies rarely include gathering information on deprecation prior to a decision. Regarding API providers, research is interested more in documentation of deprecation than in the preceding decision to deprecate.

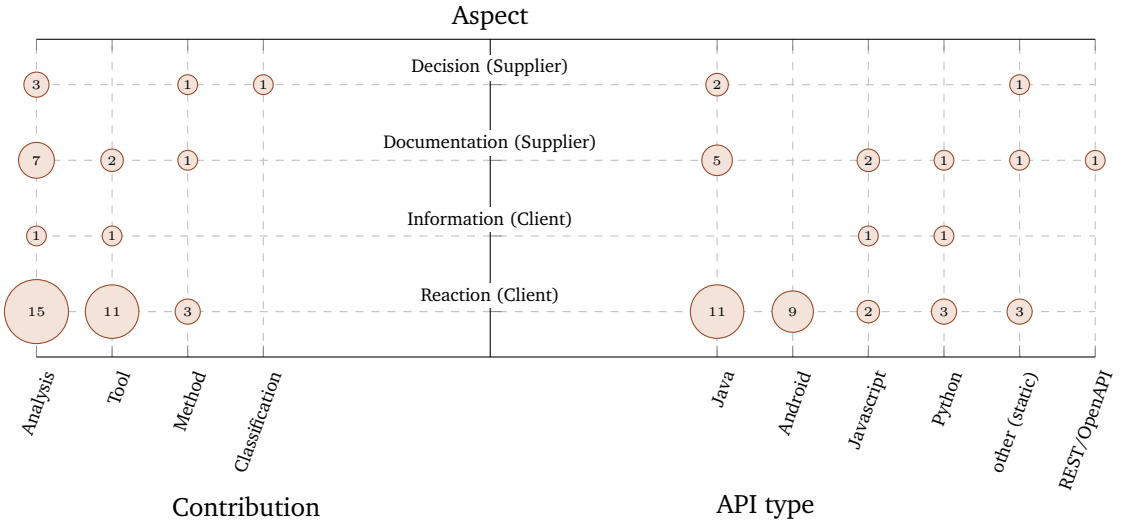

Fig. 3.8.: Map of the Research Field: A Bubble Plot Showing Aspects of Deprecation, Contributions, and Types of APIs As Considered in the 36 Primary Studies

3.3 Identification of Research Gaps

Classification and mapping primary studies reveals quantitative differences: Some areas of API deprecation are researched more than others. Since it is not a goal to study all areas with the same number of publications, we analyzed the potential gaps further by adding qualitative considerations, e.g., the relevance for pratical software engineering. This discussion leads to the following five major research gaps for API deprecation.

3.3.1 Deprecation of Remote APIs

As discussed in Sections 2.1 and 2.2, local and remote APIs as well as their evolution differ significantly. However, only Yasmin et al. (2020) investigate remote APIs, while all of the other studies research local APIs. Neither does current research describe problems and solutions for remote APIs nor does it provide constructive contributions for remote APIs.

This gap matches an observation by Raatikainen et al. (2021) from multiple case studies: Industry focus has moved from local to remote APIs, but research follows slowly.

This thesis addresses precisely this gap. In particular, the subsequent chapters analyze techniques for local APIs regarding their transferability to remote APIs. In addition, it develops new techniques for the properties specific to remote APIs.

Future research could advance these endeavors by transferring more concepts and techniques from local to remote APIs. Furthermore, it may widen the view to more types of remote APIs.

3.3.2 Diversity of Languages and Ecosystems

Regarding local APIs, research focuses on the Java programming language and the Android ecosystem – two environments with comprehensive policies and mature tool support for deprecation. While research in such a controlled and well supported context is plausible, improving deprecation processes in less mature settings would likely benefit stakeholders in practice more.

How various programming languages differ with respect to deprecation has been reported by Sawant et al. (2018a) as they identified developers' needs for deprecation support in programming languages and compared deprecation mechanisms across 21 languages: "there is no uniform manner in which languages implement their deprecation mechanism."

While this thesis develops solutions for remote APIs, it also considers local programming languages and ecosystems for the analysis of API clients. In particular, proposed solutions will not be limited to a specific programming language, but support a large variety of options. However, it is not a goal of this thesis to improve deprecation processes for local APIs.

Future research could advance deprecation mechanisms in programming languages and ecosystems. In particular, ideas from mature environments may influence advancement in less mature contexts.

3.3.3 Providers and Clients as Partners

As introduced in Section 2.2, deprecation as a process involves multiple actions by providers and clients of an API. These actions depend on each other to ensure the common goal shared by API providers and API clients: the clients' software systems should interact correctly and efficiently with the providers' software system through the API. Yet, most studies consider either providers (assuming what clients might need) or clients (accepting deprecation as indifferent force majeure) but not both.

Only four primary studies cover aspects of both, providers and clients: Nascimento et al. (2021) survey both client and provider perspectives for JavaScript, but do not compare them explicitly. Cogo et al. (2021) propose additions to JavaScript based on an analysis spanning the process from rationale of deprecation to replacement strategies.

This thesis considers actions taken by providers and clients of an API. For API providers, subsequent chapters consider a variety of options to document deprecation. Furthermore, the proposed solution allows providers to follow their preferred way of communication. For API clients, the proposed solution analyzes actual interactions and reveals relevant deprecation information. Combining both perspectives, the thesis develops a solution that allows both parties to use their preferred styles and options while enabling detection of threats to the common goal.

Future research could further join the often distinct views. In particular, descriptive studies may observe both software systems for the same deprecated element, and constructive solutions may improve transmitting and using providers' knowledge to clients and their software systems.

3.3.4 Human Aspects of Deprecation

Although many primary studies intend to benefit human stakeholders, they rarely include human perspectives but rather rely only on data sources. Only four studies use method triangulation by combining a data strategy with another empirical strategy.

This gap matches a more general observation by Storey et al. (2020) for empirical research in software engineering: Only in a few cases, improvements in data clearly improve aspects for human stakeholders. Often, insights from data require additional assessment by humans.

As an initial research contribution for deprecation of remote APIs, this thesis benefits stakeholders by providing information that has been previously not available. While the particular research does not apply research strategies involving humans

Future research could methodically triangulate insights from pure data research. This may provide additional information on the severity of problems and the applicability of solution to widen the technical view on the problem to a socio-technical consideration.

3.3.5 Prevention of Deprecation

Primary studies address all of the four steps in the deprecation model. However, the preceding root causes for deprecation are not investigated in any study. In particular, insights from the four steps are not linked back to reasons prior to the decision. Such a root cause analysis could help to prevent or predict deprecation.

This gap relates to observations by Murphy et al. (2018) as they survey API designers: “To avoid ‘breaking changes’, it is important that the API designers get core abstractions and core methods of the API correct the first time.”

This thesis does not address prevention of deprecation.

Future research could support API providers to consider likely evolution of APIs early in the initial design to make future changes easier and avoid deprecation if possible. This relates to the concepts of change prediction (Malhotra and Khanna, 2016) and fault prediction (Rathore and Kumar, 2017).

3.3.6 Summary of Research Gaps

Following the definition of these research gaps, we are now able to answer research question RQ 0.3.

Answer to RQ 0.3 *Which gaps remain in research of API deprecation?*

Research on API deprecation leaves some gaps: With respect to API diversity, studies focus on Java and Android, but should also cover remote APIs and more types of local APIs. With respect to research diversity, studies use data to study either providers or clients of an API, but should also include human insights and combine the views of providers and clients. Finally, studies omit aspects preceding the decision to deprecate, but should include those to aid API design in practice.

3.4 Discussion

3.4.1 State of Research on API Deprecation

The systematic mapping study provides insights into the state of research on API deprecation. Overall, the field shows signs of a rather young and still evolving research area: Publications focus on a few topics and only single studies analyze challenges outside of theses focuses. Furthermore, data research strategies prevail – indicating a focus on pure technical feasibility without yet considering social aspects of the technical actions. Descriptive analyses and constructive solutions are evenly balanced.

In terms of API-specific topics, the majority of primary studies investigate deprecation in the context of Java or Android. Furthermore, primary studies focus on the clients of a deprecated API and, in particular, study their reaction after they find out about the deprecation.

Classification, mapping and gap identification lead to an answer for research question RQ 0:

Answer to RQ 0 *What is the state of research on the deprecation of local and remote APIs?*

Research on API deprecation mainly addresses human stakeholders, provides descriptive analyses as well as constructive solutions, and applies data research strategies. Furthermore, research focuses Java and Android ecosystems, and concentrates on clients' reaction to deprecation. Research gaps remain for the diversity of API types (in particular, remote APIs), for an integral view on the entire process (in particular, joining views of providers and clients, and methodical triangulation) and for an extension to root causes.

3.4.2 Threads to Validity

Threats to validity for this meta-study appear in the following three categories (Ampatzoglou et al., 2019; Ampatzoglou et al., 2020; Ralph et al., 2020).

Study Selection Validity Completeness of study selection is the main threat for a systematic mapping study. To mitigate this threat, we decided to broaden the search as far as possible—narrowing the results relatively late after applying an adaptive reading strategy. This was possible since the number of identified studies was relatively low. We chose multiple databases to cover all major publication venues for software engineering research and verified the completeness by checking the inclusion of previously manually selected studies.

Inclusion and exclusion criteria were selected solely based on the study's intentions, and generic limitations (e.g., a restriction on the publication year) were not used. The decision to exclude gray literature has been motivated and complies with the study's intent. Furthermore, no sampling of relevant studies was needed as the number of results was low enough to select all results for further investigation. Decisions to select a study as an extended version of another were conducted with a consistent strategy.

Additionally, we performed forward and backward snowballing, identifying one further study. This study was published at a small workshop in 2007 and is not indexed in any major database. Thus, we conclude that this addition does not indicate a general fault in our search process.

Only one author performing the study poses a threat to validity. This researchers' bias was mitigated by discussion between the authors before each step was conducted and revision of each step's results by the second author.

Data Validity Data validity for systematic mapping studies is concerned with constructing the classification scheme and the data extraction. To mitigate this threat, we applied several strategies: We used a combination of deductive (the Who-What-How framework) and inductive scheme construction to preserve the advantages and avoid the disadvantages of both strategies. Furthermore, by using an adaptive reading strategy, no decisions have been generically limited to the contents of an abstract.

Research Validity The research process for the systematic mapping study has been described in detail. The data has been made available to ensure reliability and enable replication or extension.

3.4.3 Updates for the Deprecation of Remote APIs

The search process for the study results presented and discussed in (Bonorden and Riebisch, 2022a) has been conducted on October 1st, 2021. These results have provided the basis to define the scope of the dissertation project as discussed in Chapter 4. Nevertheless, following noteworthy newer studies on remote APIs are relevant to this thesis.

Three publications provide descriptive analyses on the deprecation of remote APIs – highlighting the need for supporting solutions further:

Di Lauro et al. (2022) survey the use of deprecation in public OpenAPI documents: "Most operations are removed without being previously deprecated." Furthermore, providers often document deprecation only in a textual description but do not use OpenAPI's deprecation flag.

Serbout and Pautasso (2024b) similarly analyze API evolution in public OpenAPI documents. They find that API developers rarely follow the rules of semantic versioning, but remove operations without deprecation and introduce other breaking changes in all types of version updates.

Lercher et al. (2024) focus remote APIs in microservice architectures and interview practitioners. While some interview partners analyze reasons for deprecation and work on avoiding deprecation (addressing the gap *Prevention of Deprecation*), a common problem is the communication with the clients of an API: "Developers do not know whom to inform, forget to notify the client teams, or convey the change information incorrectly."

Research Goal, Approach and Contribution

4

4.1 Research Goal

The systematic study in the previous chapter has identified only one research project on the deprecation of remote APIs: Yasmin et al. (2020) analyze deprecation information in OpenAPI documents and derive deprecation patterns. Yasmin (2021) applies this result further to identify calls to deprecated APIs from Javascript code. However, this approach is limited to statically available information (i. e., hard-coded URLs in the client code) and the generalization from Javascript to other settings is not trivial.

Thus, we are now able to contrast and discuss the research goal and associated research questions. Section 1.2 already presented the overall objective:

Improve	the detection of deprecated remote API usage
by	designing and implementing a framework
that	provides language-independent deprecation analysis of dynamic calls and integrates multiple data sources
in order to	enable client developers to effectively react to evolving ecosystems.

This problem statement proposes a design problem that needs to be guided by studying aspects of the problem in more detail to guide the desired design. For this, we use the following research questions:

RQ 0 What is the state of research on the deprecation of local and remote APIs?

RQ 1 How can existing static analysis techniques for local APIs be adapted to detect usage of deprecated remote APIs?

RQ 2 Which runtime properties of remote API calls are relevant to identify usage of deprecated remote APIs?

RQ 3 How can a framework be designed and implemented to improve the detection of deprecated remote API usage through language-independent deprecation analysis of dynamic calls and multi-source data integration?

RQ 4 How precise and sensitive is the analysis framework in detecting deprecated remote API usage?

Research question *RQ 0* highlights the research gap that this thesis approaches. Furthermore, it yields a collection of relevant publications on

local APIs with results that may be of relevance also for the remote setting. This preceding step has been presented in Chapter 3.

Research question *RQ 1* tries to mimic the general approach from local APIs. Furthermore, the static view relates to the preliminary work by Yasmin et al. (2020). However, this research question also aims to elaborate the limitations of purely static approaches. It guides Part II.

Research question *RQ 2* complements RQ 1 by using dynamic information and reasoning. While the goal is to overcome the limitations of static analysis, a purely dynamic approach naturally comes with its own drawbacks that need to be studied. This research question shapes Part III.

Research question *RQ 3* is an alternative formulation of the overall design objective: It focuses the construction of a solution and builds upon the preceding insights on static and dynamic approaches derived from research questions RQ 1 and RQ 2. Chapter 11 presents the final design.

Finally, research question *RQ 4* evaluates the dissertation's conceptual results quantitatively in Chapter 12 in addition to a conceptual discussion.

4.2 Approach

The general research approach follows design science research (Peffers et al., 2007; Wieringa, 2014) a practical solution is constructed and evaluated in multiple iterations. Each iteration improves the design by incorporating further understanding of the problem or feedback from the previous version.

While this method acts as a methodological framework, additional research methods are used as needed and introduced in the respective chapters. In particular, Chapter 3 has reported a systematic mapping study, Chapter 5 will feature repository mining and mixed-methods analysis, and Chapter 6 uses a search-based data science method.

The structure and presentation of this thesis does not reflect the actual iterative research process, but discusses the results in a nominal sequence as if the resulting approach and implementation had been developed in one single iteration for a concise and coherent presentation. In the actual research, previous iterations already yielded implementation prototypes – most importantly, a published version in (Bonorden and van Hoorn, 2024a).

4.3 Contributions

This dissertation project provides both, descriptive and constructive contributions as well as datasets for replication and further studies.

Descriptive empirical primary analyses include the usage of third-party HTTP clients in Java (Chapter 5, (Bonorden, 2025a)) and provision of deprecation information in OpenAPI documents (Chapter 6). Secondary analyses of publications on API deprecation (Chapter 3, (Bonorden and Riebisch, 2022a)), and relevant standardization (Chapter 9) complement the primary analyses.

Theory-building descriptive contributions lie in the systematization of the research field (Chapter 3, (Bonorden and Riebisch, 2022a)) with a four-step deprecation process model including both, API providers and clients, and a generalized three-step detection process for usage of deprecated APIs.

A constructive contribution is given by the resulting detection framework and its implementation.

Data and code publications accompany the publications (Bonorden and Riebisch, 2022b; Bonorden and van Hoorn, 2024b; Bonorden, 2025b), and the final implementation and evaluation are available as (Bonorden, 2026).

Part II

Static Deprecation Analysis

Static Analysis of HTTP Clients in Java

5

This chapter analyzes the situation of Java client code calling HTTP APIs by first investigating the use of third-party HTTP clients and subsequently studying if information about the call target is statically available. This relates to the first of the three essential steps of deprecation detection.

5.1 Third-Party HTTP Clients in Java

Identifying HTTP calls in client code in Java requires to discuss the variety of internal and third-party options for this task. The following empirical study collects 259,683 configuration files for Maven, Gradle, and Ant/Ivy build tools from 18,879 open-source Java repositories, and subsequently analyzes the use of third-party HTTP clients quantitatively and qualitatively. This acts as a preliminary for the actual detection of individual calls in Section 5.2.

Publication Note This chapter comprises the study *"An Empirical Analysis on the Use of Third-Party HTTP Clients in Open-Source Java Projects"* published in (Bonorden, 2025a). The artifact (Bonorden, 2025b) presents supplementary material.

5.1.1 HTTP Clients in Java

For the Java programming language, there is not a single definitive way to implement an HTTP call, but a variety of alternatives. In addition to two internal options (`HttpURLConnection` and `HttpClient`), a variety of third-party client libraries exist.

HttpURLConnection Historically, Java 1.0 (released in 1996) featured a fundamental networking library `java.net` including a class `URLConnection` that was designed to represent an active connection. This design allowed to define an individual `URLStreamHandler` for the protocol used. The language designers expected a variety of specialized use cases, e. g., `http://` for the world wide web, `gopher://` for Gopher (an alternative to the world wide web), and `news://` for Usenet newsreaders (JEP 321, 2017; Berners-Lee et al., 1994).

Subsequently, Java 1.1 (released in 1997) added `HttpURLConnection` as a subclass to *URLConnection* to represent a connection using HTTP as protocol without the need to implement a custom `URLStreamHandler`. Subsequently, Java 1.2 added `JarURLConnection` as an additional subclasses for Java Archive Files. Furthermore, Java 1.4 added `HttpsURLConnection` as a subclass to `HttpURLConnection` for the HTTPS protocol. However, the

`HttpURLConnection` also handles HTTPS connections, but does not allow to use customized SSL settings or to inspect SSL certificates.

The `HttpURLConnection` represents data in HTTP requests and responses as streams. This allows to read messages line by line, but requires to open and close connections manually. To read the entire response of an HTTP call as one single object, it is necessary to read all lines individually and then concatenate them as shown in Listing 5.1.

Java

```
1  URL url =
2      new URL("https://time.example/timeservice?placeid=germany/hamburg");
3  HttpURLConnection connection = (HttpURLConnection) url.openConnection();
4  connection.setRequestMethod("GET");
5
6  StringBuilder body = new StringBuilder();
7  try (BufferedReader reader = new BufferedReader(
8          new InputStreamReader(connection.getInputStream(),
9              StandardCharsets.UTF_8))) {
10     String line;
11     while ((line = reader.readLine()) != null) {
12         body.append(line);
13     }
14 }
15
16 LocalDateTime result = LocalDateTime.parse(body.toString());
```

Listing 5.1.: Example Code to Call the Time API using `HttpURLConnection`.

The characteristics of the `HttpURLConnection` show in lines 3 (opening a connection) and 8 (capturing the response as an input stream with an `InputStreamReader`).

The example is simplified: The code does not check the response code prior to reading the response body. Furthermore, the code does not close the connection in the end. Finally, the code assumes a correctly formatted response body and does not check for errors.

Third-Party HTTP Clients To overcome the technical nature of the build-in `HttpURLConnection`, third-party alternatives arose. One of the first external HTTP libraries was the *HttpClient* in the Jakarta Commons project – started in 2001. In 2007, the Commons project moved to the Apache Software Foundation and the client became known as *Apache HttpClient* as part of *Apache HttpComponents*.

Since then, a variety of further third-party clients emerged with different feature sets, usage styles or performance results. This chapter will

later address this diversity and show example code in different styles in Section 5.2.2.

To include a third-party library, developers typically use build tools (e. g., Maven or Gradle) that import such dependencies from artifact repositories.

Java HttpClient Following the success of third-party libraries, the Java designers acknowledged the shortcomings of *HttpURLConnection* and created a new *Java HttpClient* as a replacement (JEP 110, 2014). Following an "incubator" phase, the client was released with Java 11[1] in 2018 (JEP 321, 2017). The development for the *Java HttpClient* remains active and recently added support for HTTP version 3 (JEP 517, 2022).

To distinguish the *Java HttpClient* from other HttpClients in Java, the alternative name *JDK HttpClient*, its full package name `java.net.http.HttpClient`, or an abbreviated form *java.net HttpClient* are sometimes used.

The *Java HttpClient* explicitly models the client, the request and the response as individual objects. In contrast to the `HttpURLConnection`, opening and closing connections is no longer a manual task, and responses are a single unit instead of line-by-line streaming. Popular HTTP Clients, e. g., the Apache HttpClient, inspired the design of these features in the *Java HttpClient*. Listing 5.2 shows a code example.

Java

```
1 URI uri =
2     URI.create("https://time.example/timeservice?placeid=germany/hamburg");
3 HttpClient client = HttpClient.newHttpClient();
4 HttpRequest request = HttpRequest.newBuilder(uri).build();
5 HttpResponse<String> response =
6     client.send(request, BodyHandlers.ofString());
7 LocalDateTime result = LocalDateTime.parse(response.body());
```

Listing 5.2.: Example Code to Call the Time API using the *Java HttpClient*.

The characteristics of the *Java HttpClient* show in lines 3–6: client and request are individual objects. After instantiation, the client can send the request to receive a response object.

The example is simplified: The code does not check the response code or the response body for errors, but assumes them to be successful and correct.

[1] While early Java versions were numbered 1.x, the release *Java 1.5* was changed to *Java 5* and subsequent releases are denoted without the "1." prefix.

Choosing an HTTP Client Nowadays, internal and external HTTP clients are close to each other in terms of features (e. g., persistent connections) and non-functional quality criteria (e. g., performance). Small differences appear only in special usage scenarios (e. g., automatic GZIP compression for requests) or with respect to the timely support for new HTTP features (e. g., HTTP version 3).

New projects may choose a third-party client if it features unique advantages, or opt for the internal Java HttpClient to reduce external dependencies. Existing projects may keep their current client to minimize migration effort or migrate to the internal Java HttpClient.

5.1.2 Research Design

Research Objective While it is clear, that a deprecation detection framework needs to support both internal options, the choice of third-party clients to support requires a data-based decision. Thus, this study aims to analyze the state of third-party HTTP clients in Java broadly. In particular, the study pursues quantitative information about clients and their versions as well as qualitative insights into the actual use of these clients. These objectives motivate the following research sub-questions for research question RQ 1:

RQ 1 **How can existing static analysis techniques for local APIs be adapted to detect usage of deprecated remote APIs?**

RQ 1.1 Which HTTP clients and versions do open-source Java projects use?

RQ 1.2 How do open-source Java projects use HTTP clients?

Chapter 6 will add another research sub-question to RQ 1 to also study API documentation on the provider's side – i. e., step 2 of the essential steps for deprecation detection.

Related Work The related work for this study originates from several perspectives on the issue. First, **architecture evaluation and reconstruction** may utilize HTTP/REST communication: Jasser (2020) specified security rules like "No ThirdPartyBuildingBlock can send a Message that (leaves the System)" and utilized HTTP clients to enforce them. Genfer and Zdun (2021) and Hutcheson et al. (2024) detected HTTP/REST communication statically to extract communication structures in microservice environments. Such approaches depend on the identification of actual HTTP/REST communication and rely on individual detectors for each HTTP client –

often implemented for the specific use case. In contrast, this study investigates the situation of third-party HTTP clients fundamentally and operates on a broader project level.

Another perspective arises from the **Android ecosystem**: Rapoport et al. (2017) identified the call targets of Android apps both statically and dynamically. They found that a large number of dynamic requests did not use a statically detectable URL. Other studies investigated usage numbers of HTTP clients in Android apps: Gadient et al. (2020) manually studied 160 apps and found *Retrofit* and *OkHttp* to be the most used third-party HTTP libraries. Abdellatif et al. (2020) analyzed 1,595 Android apps automatically and identified *Retrofit*, *OkHttp* and *Google Volley* as the most popularly used third-party HTTP libraries. In 2018, 98% of newly released apps used a third-party HTTP library. The transferability of these results from Android to general Java projects is unclear. Thus, this study aims to provide similar insights for the general scope.

Focusing the **static build dependencies** necessary to use third-party libraries yields another perspective, but most studies focus dependencies between Maven artifacts. Yoshioka et al. (2025) observed that deprecated versions remain popular and are even used by new projects. Huang et al. (2022) confirmed the use of outdated versions and further found that the number of external libraries grows with the project size. Soto-Valero et al. (2021a) and Soto-Valero et al. (2021b) found that transitively included dependencies (dependencies of the declared dependencies) often add external libraries that are never used. Suwanachote et al. (2025) discovered that about half of all Maven projects on GitHub declare unused dependencies, and Keshani et al. (2024) described "exponential dependency growth" for such transitive dependencies. On the contrary, Harrand et al. (2021) and Riegler et al. (2023) highlight positive aspects of library diversity, e. g., for switching between two similar libraries. Such studies investigate the Maven ecosystem in general. This study adds a more in-depth perspective for the special case of HTTP clients.

Finally, dependencies are also relevant for a **software bill of materials (SBOM)**. SBOMs feature an alternative way to list dependencies and are likewise used by software architecture research, e. g., Speth et al. (2025). While Gamage et al. (2025) find that SBOMs in Maven central are generally well aligned with the projects build dependencies, Xiao et al. (2025) report that 30% of Java projects on GitHub miss direct dependencies in their SBOMs. Thus, we instead focus on configuration files for this study and do not use SBOMs.

Research Method Repository Mining (Ralph et al., 2020; Vidoni, 2022) allows to extract a dataset from a hosting platform to perform subsequent

data analysis. The data extraction for this study filters GitHub repositories using the *SEART GitHub Search Engine* (Dabic et al., 2021) and the following filter criteria:

1. Java is the repository's main language.
2. The repository has at least 10 stars.
3. At least two contributors provided commits to the repository.
4. The repository's latest commit was after 2024-04-01.

While the first filter ensures the scope of HTTP clients in Java, the second and third filter add moderate restrictions to filter out low-quality and personal projects. The fourth filter ensures that only recently active projects (with respect to the final query execution on 2025-03-08) contribute to the analysis. This selection procedure yields 18,879 repositories.

Identification of HTTP communication spans options on different abstraction levels, i. e., checking either a statement, a method, a class or the entire project for indicators. While identification on statement or method level is more detailed, in particular regarding call targets, a broader declaration of dependencies on the broader project level is a necessary (though not sufficient) condition for HTTP communication. The broader level allows to use configuration files for identification – as opposed to source code or its derivatives. However, identification via static dependencies cannot distinguish between multiple parts of an artifact, e. g., if a HTTP client from the artifact is actually used, or which of multiple clients within the same artifact are used.

A software tool, the `DependencyCollector` (part of the replication package (Bonorden, 2025b)), now traverses the identified repositories and fetches configuration files via the GitHub API. These files include configurations for the three major Java build tools: *Maven*, *Gradle*, and *Ant* (Sulír et al., 2026). While Maven and Gradle directly include dependency management, Ant utilizes a dedicated dependency manager *Ivy*. The results comprise 259,683 configuration files in 17.913 repositories:

- 24,847 repositories (60.1 %) feature Maven configuration files, i. e., `pom.xml` files.
- 15,979 repositories (38.6 %) feature Gradle configuration files, i. e., `build.gradle` or `build.gradle.kts` files.
- 510 repositories (1.2 %) feature Ant/Ivy configuration files, i. e., `ivy.xml` files.
- 966 repositories (5.1 %) do not feature any of the three build tools.

All three dependency managers may utilize different artifact repositories, but mainly use the *Maven Central Repository*.

To check these configurations for the inclusion of HTTP clients, a list of artifacts containing HTTP clients is necessary. We manually compile such a list of HTTP clients and their corresponding 146 artifacts in the Maven Central Repostiory from three sources:

- **Scientific publications.** We search Google Scholar for "Java HTTP clients" and for combinations of known clients (e.g., "Java okhttp apache jetty").
- **MvnRepository.** Similar to the official Maven repository, the website mvnrepository.com lists Maven artifacts, but adds categories like "HTTP Clients"[2].
- **Grey Literature.** We searched Google for "Java HTTP client" and checked the first 100 results manually.

Progressing with the analysis requires trade-off decisions to balance depth and breadth. We decide to split the further analysis into three parts.

For a **broad quantitative analysis** without sampling, i. e., using all of the 17,913 repositories with configuration files, the second tool, `Dependency-Identifier` (also part of the replication package (Bonorden, 2025b)), statically compares the declared dependencies against the compiled list of known HTTP client artifacts. The statical nature of the analysis (i. e., not querying the Maven Central repository) allows for the broadest possible investigation, but excludes properties that would only be calculated dynamically (e. g., listing transitive dependencies and resolving version variables like "latest"). This part of the analysis yields the insights for Section 5.1.3.

For an **in-depth quantitative analysis** using random sampling (Baltes and Ralph, 2022), we select a subset of 100 repositories. Again, the `DependencyIdentifier` scans the projects for HTTP client artifacts, but this time actually querying the Maven Central repository, to include transitive dependencies and version information. These results contribute to Section 5.1.4.

For a **qualitative manual analysis**, we filter for repositories using one of the three major HTTP client artifacts. This selection results from the first part of the analysis: Apache HttpClient, Spring Web and OkHttp. Subsequently, we randomly select (Baltes and Ralph, 2022) 10 repositories for each of the three client artifacts for an in-depth analysis: (1) We check if the HTTP client from the declared dependency is actually used. (2) We check if multiple clients (internal or third-party) are used. (3) We check how call targets are passed to the HTTP clients – in particular, if the

[2] mvnrepository.com/open-source/http-clients

exact information about the call target is provided statically. Section 5.1.5 presents the results for this analysis.

5.1.3 Quantitative Insights on the Usage of Third-Party HTTP Clients

Approximately half of the repositories with configuration files (51 %) declare a direct dependency on an HTTP client artifact. Out of these repositories with such a dependency, 3741 (40 %) include the Apache HttpClient, 1893 (20 %) include Spring Web clients, and 1606 (17 %) include the OkHttp client. Fig. 5.1 shows the numbers of occurrence for all HTTP client artifacts that were observed at least 100 times.

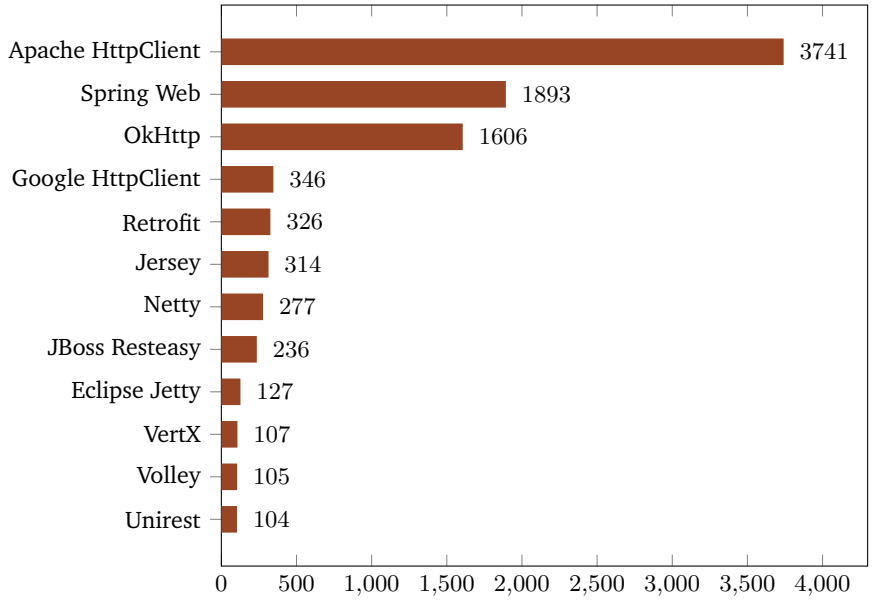

Fig. 5.1.: Occurrences of direct dependencies to popular third-party Java HTTP clients in 18,879 GitHub repositories

Since the three most popular HTTP client artifacts account for approximately 80 % of all direct dependencies, we focus on those. The following paragraphs provide a short introduction and the clients, and Section 5.2 will discuss their usage in more detail. The supplementary material (Bonorden, 2025b) provides additional information – including legacy information and additional artifact names.

The **Apache HttpClient** is part of the open-source Apache HttpComponents project and comprises a low-level framework for custom clients ("HttpCore" with the Maven artifact `org.apache.httpcomponents.core5:httpcore5`)

and a ready-to-use HTTP client for standard use cases (“HttpClient” with the Maven artifact `org.apache.httpcomponents.client5:httpclient5`). Due to the low-level configurability, this client can efficiently handle high performance demand.

The open-source **Spring Web** framework features a single Maven artifact `org.springframework:spring-web` that comprises three distinct HTTP clients: RestClient, WebClient, and RestTemplate. Despite their names, they all offer features for arbitrary HTTP communication. The *RestClient* and the *WebClient* are abstractions over multiple HTTP libraries and use another HTTP client internally, e. g., the Apache HttpClient or the Java HttpClient. While the *RestClient* is a general purpose option, the *WebClient* offers features specialized for reactive programming. The *RestTemplate* is a deprecated legacy option. Since these HTTP clients are part of the Spring Web artifact, they are a common choice for projects using the Spring Web framework. Contrarily, they are inconvenient for projects without the Spring Web framework.

The **OkHttp** client is an open-source project featured in the Maven artifact `com.squareup.okhttp3:okhttp`. It is backed by the card payment and point-of-sale service provider Square. In addition to its general purpose feature, this client integrates well with the Android operating system and the Android ecosystem (Gadient et al., 2020).

Java

```
1 CloseableHttpClient client = HttpClients.createDefault()
2 HttpGet request = new HttpGet(
3     "https://time.example/timeservice?placeid=germany/hamburg");
4
5 CloseableHttpResponse response = client.execute(request);
6 String body = EntityUtils.toString(response.getEntity(),
7     StandardCharsets.UTF_8);
8 LocalDateTime result = LocalDateTime.parse(body);
```

Listing 5.3.: A Call to the Time API using the *Apache HttpClient*.

The characteristics of the *Apache HttpClient* are similar to the *Java HttpClient*: explicit objects for client, request, and response.

The example is simplified: The code does not check the response code or the response body for errors, but assumes them to be successful and correct.

```java
RestClient client = RestClient.create();

String body = client.get()
        .uri("https://time.example/timeservice?placeid=germany/hamburg")
        .retrieve()
        .body(String.class);

LocalDateTime result = LocalDateTime.parse(body);
```

Listing 5.4.: Call to the time API using the *Spring RestClient*.

Client and response are similar to other clients, but no explicitly modeled request is used.

The example is simplified: The code does not check the response code or the response body for errors, but assumes them to be successful and correct.

```java
OkHttpClient client = new OkHttpClient();
Request request = new Request.Builder()
    .url("https://time.example/timeservice?placeid=germany/hamburg")
    .build();

Response response = client.newCall(request).execute();
String body = response.body().string();
LocalDateTime result = LocalDateTime.parse(body);
```

Listing 5.5.: Call to the time API using the *OkHttp Client*.

The code uses explicit client, request and response objects. This is similar to the *Java HttpClient* and the *Apache HttpClient*.

The example is simplified: The code does not check the response code or the response body for errors, but assumes them to be successful and correct.

5.1.4 Quantitative Insights on Transitive Dependencies and Artifact Versions

Including transitive dependencies in the analysis, the number of clients per project rises significantly: a project includes 2.1 clients transitively on average, but in some instances a single project included up to 22 HTTP clients transitively.

Adding version information for the three major HTTP clients, we can classify third-party client usage into four categories with respect to the cut-off-date 2024-01-01:

- **Up-to-date.** The project updated to a new version within 3 months of release, i. e., the version used was not yet superseded or only after 2023-10-01. We call these versions *up-to-date* as they show timely reaction to new releases.
- **Acceptable.** The project updated to a new version within 12 months of release or after the publication of a security issue, i. e., the version used is not related to a published CVE (i. e., a security issue recorded in the*Common Vulnerabilities and Exposures* system) and was not yet superseded or only after 2023-01-01. We call these versions *acceptable*, because there is no strong indication against their use.
- **Outdated.** The project did not update to a newer version within 12 months of release or after the publication of a security issue, i. e., the version used is related to a published CVE or was superseded prior to 2023-01-01. We call these versions *outdated*, because there is strong indication to update to a newer version.
- **Antique.** The project missed out on at least two major version updates, i. e., the major version used was at least by 2 lower than the newest released version on 2024-01-01. If this category applies, the classification ignores the other options. We call these versions *antique* as they are signs of a missing dependency management.

Due to the static nature of these dependencies and their inclusion in the build process, older versions still provide their original functionality. However, the maintenance often ends after the release of a new version and newer features are not added to older versions.

Examples for outdated versions subject to security issues are versions of Apache HttpClient prior to 4.5.13 and versions of OkHttp prior to 4.1.12. The versions of the Apache HttpClient are affected by CVE-2020-13956[3], which allows attackers to use malformed parameters to alter the call target. The versions of the OkHttp client are affected by CVE-2023-3635[4], which allows a denial-of-service attack using malformed buffers.

Overall, the evaluation finds approximately half of the versions used to be up-to-date or acceptable (48 %). Distinguishing the three major HTTP clients, different situation arise:

Projects with the **Apache HttpClient** often use acceptable or outdated version, but only rarely up-to-date or antique versions. This indicates a general consciousness for dependency management and strong appeal for new major versions, e. g., by significant improvements.

[3] www.cve.org/CVERecord?id=CVE-2020-13956, issued in October 2020
[4] www.cve.org/CVERecord?id=CVE-2023-3635, issued in October 2023

For the **Spring Web** dependencies, projects use up-to-date, outdated and antique versions equally, but no observation of acceptable version occurs. This indicates a general aversion to updates, but sticking with the current version if possible.

Finally, **OkHttp** projects favor up-to-date or antique versions, but rarely use acceptable or outdated versions. This indicates a general willingness for swift updates to new minor versions, but also an aversion of updates to new major versions.

Table 5.1 gives an overview of these version classifications.

Tab. 5.1.: Classification of versions used for popular third-party HTTP clients (observation from 100 randomly sampled projects

Classification	Apache HttpClient	Spring Web	OkHttp	Total
up-to-date (< 3 months)	23 (4 %)	5 (28 %)	13 (35 %)	41 (6 %)
acceptable (< 12 months)	289 (45 %)	—	7 (19 %)	296 (43 %)
outdated (≥ 12 months)	318 (50 %)	7 (39 %)	6 (16 %)	331 (48 %)
antique (≥ 2 major versions)	12 (2 %)	6 (33 %)	11 (30 %)	29 (4 %)
Total	642	18	37	697

5.1.5 Qualitative Characterization of Third-Party HTTP Client Usage

With the manual qualitative analysis, we characterize recurring patterns in the context of third-party HTTP clients.

Unnecessary Dependencies If the build process includes a dependency that the software never uses, such dependency is called "bloated" (Soto-Valero et al., 2021b). This commonly occurs if dependencies are included transitively – often unknown to the developers adding the direct dependency. However, we observed in 6 out of 30 repositories (20 %) that a directly declared dependency to a HTTP client was never used in the project's source code.

Furthermore, 7 out of 30 repositories (23 %) featured only a single HTTP call. In none of these cases could we identify a reason not to use an internal option, but to add a build dependency.

Internal HTTP Clients Although the study focuses third-party HTTP clients, we also looked for use of internal clients. Only 1 out of 30 repositories (3 %) uses the Java HttpClient, but 9 out of 30 repositories (30 %) use `HttpURLConnection`.

Multiple HTTP Clients The combination of a third-party library and an internal client occurs in 7 out of 30 repositories (23 %). In none of the cases we could justify the need for two clients.

However, we found combinations of multiple third-party clients appropriate. In the instances we observed, the projects either included plugins for many HTTP clients or included tests with multiple HTTP clients.

Provision of Call Targets In 2 out of 30 repositories (7 %), we found the call targets in the source code or configuration files within the repository. In all other cases, the call targets were determined only dynamically at runtime. Section 5.2.3 discusses call target definition further.

5.1.6 Discussion

The plethora of HTTP clients for Java is not reflected by the actual use in open-source Java repositories. Instead, the majority of software projects uses one of three major HTTP clients. This reduces the number of HTTP clients that are indispensable for further analysis.

However, even active repositories do not manage their dependencies to HTTP clients effectively. The study repeatedly found unused dependencies, small use cases that do not require an external dependency, and use of outdated versions. Thus, further analysis needs to take this diversity within the usage of the three major HTTP clients into account.

Furthermore, we are now able to answer the first two sub-questions for research question RQ 1:

Answer to RQ 1.1 *Which HTTP clients and versions do open-source Java projects use?*

Open-source Java projects mainly use the Apache HttpClient, the Spring Web clients, and the OkHttp client. Other third-party clients are used significantly less. The projects use a variety of different versions, including versions that are two or more major versions behind the current release.

Answer to RQ 1.2 *How do open-source Java projects use HTTP clients?*

Open-source Java projects do not use third-party HTTP clients in a uniform way. No clear reasons for the usage of a certain third-party or internal HTTP client arose. Instead projects often appear to include third-party HTTP clients without further consideration or subsequent management of these dependencies. The actual calls performed with the clients are often not completely analyzable statically without executing the program. In particular, the call targets or parts of the call targets are not given in the source code or configuration files.

Recommendations The study immediately motivates recommendations regarding third-party HTTP clients: For practitioners, we recommend more attention to an active dependency management. This includes reviewing and updating existing dependencies as well as caution with the addition of new dependencies when internal options may suffice. For researchers studying HTTP clients further, we recommend to expect more than one HTTP client per project and to focus on the three major HTTP clients.

Threats to Validity By design, the study features natural **limitations**: It only considers open-source projects, identifies usage of third-party HTTP clients on artifact-level and thus cannot distinguish between multiple clients within the same artifact, and does not consider internal options.

Selecting repositories and the data to include in the study poses a threat to its **external validity**. Significant elements may be missed and conclusions might be not generalizable to other repositories. We mitigated this threat

by consulting guidelines and common processes for repository mining Vidoni, 2022; Ralph et al., 2020.

To maximize **internal validity**, we avoided sampling if possible, and if not avoidable, applied random sampling to prevent bias. Furthermore, we combined quantitative and qualitative approaches to ensure a consistent interpretation. However, the internal validity might still be threatened if the automated code performs inaccurately.

Construct validity is supported by careful consideration of under-representation and over-representation of concepts. To avoid under-representation, concepts are taken as broad as possible, e. g., each artifact enabling a HTTP call is considered a HTTP client. To avoid over-representation, manual analysis checks the results from automated quantitative analysis. However, constructs might be not represented accurately, e. g., if important properties like decision rationale can not be identified from the data alone.

We defined all search, filter and sampling criteria a priori. Furthermore, we improve **conclusion validity** by making reasoning from the results to the discussion explicit. Yet, parts of the analysis were performed on a smaller subset of repositories affecting the internal validity.

5.1.7 Conclusions

The empirical study shows a majority of projects using only few different clients, but in many different versions. The remaining minority is spread across numerous, but rarely used clients. A few popular styles and usage patterns appear, but there is no definitive format for HTTP calls in Java. Most importantly, developers can store or construct the call target in many different ways.

For the detection of calls to deprecated HTTP APIs, this is a major limitation. The generalized three-step deprecation detection process builds on a correct and complete detection in the first step. The subsequent Section 5.2 will discuss detection strategies in more detail, but if information about the call target is missing, no detection will be able to determine it.

5.2 Static HTTP Call Identification

Following the analysis of HTTP calls in Java on a client level, this section further progresses towards an answer for RQ 1: How can existing static analysis techniques for local APIs be adapted to detect usage of deprecated remote APIs? For this, the static detection of an actual call within the source code together with all relevant information is the next topic of interest.

5.2.1 Related Work

Rapoport et al. (2017) and Gadient et al. (2020) analyzed HTTP/REST calls in Android applications. Both searched for URL strings to locate these external calls but Rapoport et al. reported low agreement between static and dynamic analysis and the study by Gadient et al. has low precision (0.46) and recall (0.80). While the idea of searching for URL strings fulfills our goal of an analysis solely based on the source code, it does not cover the entirety of external communication forms and is restricted to clients with statically extractable URLs, excluding targets that are determined at runtime or via external configuration files.

Rademacher et al. (2020) presented a modeling method for the reconstruction of microservice architectures and detailed it for Java-based microservices implemented with the Spring Framework and Docker container technology. In a series of steps, the method comprises analyses of the domain, the services themselves, and operations between them. The connections between the microservices are identified via Docker configuration files. However, these results are not generalizable beyond a well-known microservice architecture.

With the goal of security analysis, Schneider and Scandariato (2023) also studied Docker files, but additionally included other configuration files. Furthermore, they applied a keyword search, but restricted it to unambiguous keywords, e. g., `"@OpenFeign"` or `"@EnableAuthorizationServer"`. This direct search fails if ambiguous methods are called, e. g., an `execute` method for an *OkHttp* client. A similar direct search for unambiguous annotation names (of Spring clients) has been used by Ma et al. Ma et al. (2019).

Without the need for additional Docker or other configuration files, Pigazzini et al. (2020) identified calls using the *Feign* client or the *Spring RestTemplate* and extracted information about the targets called. However, these results are limited as they are very precisely constructed for a specific setting and do not generalize to arbitrary HTTP calls.

5.2.2 Calls in HTTP Clients

To prepare the actual detection of HTTP calls in Java code, understanding how these calls appear with different HTTP clients is an essential prerequisite. Manually analyzing the list of HTTP clients and actual client code for these libraries yields the following generalized patterns of HTTP client usage. Each pattern is named after the explicitly available objects it comprises.

Client-Request-Response This structure occurs most often in HTTP clients and uses three explicit steps: creating or obtaining a client object; creating a request object; passing the request object to the client object to yield a response object. Listing 5.6 shows the general setup.

Java

```
HttpClient client = new HttpClient();
Request request = new GetRequest(
    "https://time.example/timeservice?placeid=germany/hamburg");
Response response = client.execute(request);

String body = response.getBody();
LocalDateTime result = LocalDateTime.parse(body);
```

Listing 5.6.: The Client-Request-Response Pattern for HTTP Clients in Java.

In this pattern, client, request and response use three explicit objects: The *client* executes a *request* to yield a *response*.

This example code is abstracted from multiple client libraries and does not relate directly to one of the libraries.

Among others, this structure is supported for the internal Java HttpClient and external clients Apache HttpClient, OkHttp, Google HttpClient, Jersey, RESTEasy, and Unirest.

Client-Response In contrast, this pattern does not use an explicit request object, but executes a method on the HTTP client object instead. Listing 5.7 shows the abstracted code.

Java

```
HttpClient client = new HttpClient();
Response response = client.get(
    "https://time.example/timeservice?placeid=germany/hamburg");

String body = response.getBody();
LocalDateTime result = LocalDateTime.parse(body);
```

Listing 5.7.: The Client-Response Pattern for HTTP Clients in Java.

In this pattern, client and response use explicit objects, but no explicit request is given: The method to yield the response is called directly on the client object.

This example code is abstracted from multiple client libraries and does not relate directly to one of the libraries.

Spring RestClient, Spring RestTemplate, Spring WebClient, Jersey (Fluent API), Netty HTTP Client, and Jetty HttpClient use this pattern.

Request-Response Similarly, this pattern also combines client and request into one object – but a request instead of a client. Listing 5.8 shows this version.

Java

```
Request request = new GetRequest(
    "https://time.example/timeservice?placeid=germany/hamburg");
Response response = request.execute();

String body = response.getBody();
LocalDateTime result = LocalDateTime.parse(body);
```

Listing 5.8.: The Request-Response Pattern for HTTP Clients in Java.

In this pattern, request and response use explicit objects, but no explicit client is used: The request to yield the reponse is executed without the provision of an explicit client object.

This example code is abstracted from multiple client libraries and does not relate directly to one of the libraries.

The clients Apache HttpClient (Fluent API), Google HttpClient, Unirest, and Vert.X HttpClient apply this structure directly. Furthermore, Volley uses a variation of the pattern by collecting multiple requests in a `RequestQueue`.

API Interface Contrasting the three similar options discussed before, this pattern follows a distinct idea: providing a Java class API instead of using individual HTTP request and response messages. The provided Java class API mimics the appearance of local Java APIs.

To achieve this, an interface for the general API provides Java methods for each individual operation. Annotations define the mapping between interface methods and HTTP requests as shown in Listing 5.9.

Java

```java
interface TimeService{
    @GET("/timeservice?placeid={placeid}")
    String getTime(@Param("placeid") String placeId);
}

TimeService timeservice =
    HttpClient.build(TimeService.class, "https://time.example");
String responseBody = timeservice.getTime("germany/hamburg");

LocalDateTime result = LocalDateTime.parse(responseBody);
```

Listing 5.9.: API Interface Pattern for HTTP Clients in Java.

In this pattern, API endpoints are modeled as Java interfaces (lines 1–4). An instance of the interface is constructed by the HTTP library if necessary (lines 6-7).

This example code is abstracted from multiple client libraries and does not relate directly to one of the libraries.

The clients Retrofit, RESTEasy, and OpenFeign use this pattern.

Connection Again in contrast to all previously discussed patterns, Listing 5.10 shows a lower-level connection-focused option. As shown in Listing 5.10, the client code establishes a connection to an HTTP endpoint and the response is then read via a stream. To obtain the response as a single object, manual collection of individual response parts is necessary.

Java

```java
Connection connection = new Connection(
    "https://time.example/timeservice?placeid=germany/hamburg");
connection.setMethod("GET");

InputStream stream = connection.getInputStream();
String body = stream.readAll();

LocalDateTime result = LocalDateTime.parse(body);
```

Listing 5.10.: Connection Pattern for HTTP Clients in Java.

In this pattern, the communication via the API is modeled with streams. The client may send and receive transmissions through the stream, or record the information to yield an object for the entire reponse.

This example code is abstracted from multiple client libraries and does not relate directly to one of the libraries.

The Java class HTTPUrlConnection follows this idea. In addition, Netty provides an option to operate in a similar low-level pattern.

Asynchronous Client–Response Finally, an asynchronous usage option completes the list of patterns: The pattern is similar to Client–Response, but does not wait for a response to continue. Instead, the response is equipped with instructions to execute as soon as the reponse is received.

Java

```
HttpClient client = new HttpClient();
AsyncResponse response = client.getAsync(
    "https://time.example/timeservice?placeid=germany/hamburg");

LocalDateTime result;
response.onSuccess( body -> {
    result = LocalDateTime.parse(body);
}
```

Listing 5.11.: Asynchronous Client–Response Pattern for HTTP Clients in Java

Similar to the (synchronous) client–response pattern, the explicit client is instructed to perform a request yielding a response. However, this response is a wrapper to receive the actual content later once the answer is received asynchronously. The response is instructed on how it should proceed when the answer arrives.

This example code is abstracted from multiple client libraries and does not relate directly to one of the libraries.

The HTTP clients Spring WebClient and Vert.x Web Client follow this pattern. Furthermore Apache HttpClient offers an HttpAsyncClient as a variation to allow this usage.

Other HTTP clients offer asynchronous patterns that do not follow this pattern strictly. Since they are individual to the respective clients, we do not define another asynchronous pattern.

5.2.3 Target Definitions for HTTP Calls

Each of the HTTP clients provide a well-defined option to provide the call target. The call patterns in Section 5.2.2 show such target provision through method parameters (as a String or as a dedicated URI object) or in the annotations of an interface-based pattern. In some cases, the provision is split across multiple parts, e. g., a base URI and a specific operation.

However, the location of the actual call target definition varies greatly throughout the observed examples. This is independent of the client or call pattern, and we observed another set of patterns:

Hard-coded with the Call The easiest option for static source code analysis is the case of a hard-coded URL within the respective method call. The pattern code listings use this example, and Listing 5.12 repeats the relevant parts.

Java

```
Request request = new GetRequest(
    "https://time.example/timeservice?placeid=germany/hamburg");
Response response = request.execute();
```

Listing 5.12.: A Hard-coded Target in the Call Definition.

In the easiest case for static code analysis, the URL is provided as hardcoded parameter information.

Provided in Source Code Another variation provides the URL in the project's source code, but not directly where it is passed to the HTTP client. This type of location comprises local variables and class variables in the same class as well as hard-coded information in other classes. Listing 5.13 provides an example.

Java

```
private final String TIME_API =
    "https://time.example/timeservice?placeid=germany/hamburg";

Request request = new GetRequest(TIME_API);
Response response = request.execute();
```

Listing 5.13.: A Target Provided in the Source Code.

The call target is provided via a variable. The complexity to obtain the call target statically depend on the local of the target definition in the source code. While a definition within the same method is usually detectable, other locations in the source code are challenging.

Provided in Configuration Files Multiple frameworks allow the definition of properties in dedicated configuration files. These files are either provided by the framework itself (e. g., `application.properties` for Spring) or by another tool (e. g., `pom.xml` for Maven configuration). Listing 5.14 show a property definition via Maven.

Maven Configuration (XML)

```xml
<properties>
    <TimeAPI>
        https://time.example/timeservice?placeid=germany/hamburg
    </TimeAPI>
</properties>
```

Java

```java
Request request = new GetRequest(properties.getTimeAPI());
Response response = request.execute();
```

Listing 5.14.: A Target Provided in a Maven Configuration File.

The call target is not provided *in* the source code, but together with the source code. This requieres additional tooling that is specifically able to analyze the concrete type of addition configuration file.

Not Provided Finally, the program may read the call target only dynamically, e. g., from user input or form a previous request. In these cases, the call target is not provided with the static source files. Listing 5.15 shows such an approach.

Java

```java
List<URI> fileURIs = getFilesToDownload();
List<Reponse> responses = new ArrayList<>();
for (URI uri : fileURIs) {
    Request request = new GetRequest(uri);
    responses.add( request.execute() );
}
```

Listing 5.15.: A Target that is only received Dynamically.

The target is obtained at runtime, possibly as the result of another call. As the static analysis only covers the client's code, static identification is usually not possible.

5.2.4 Detection Methods

Detection of HTTP calls in general or of call targets specifically is possible on multiple abstraction levels (e. g., source code or abstract syntax trees) and in multiple steps of a build process (e. g., raw source code or after lexical analysis).

Furthermore, the detection itself may follow a multi-step process, e. g., first identifying candidate locations and subsequently investigating in more detail.

Following the definition of call patterns and options for target provision, we briefly discuss the following approaches to HTTP call identification.

Text Search Without any additional information or processing steps, source code takes the form of text files. For these, text search is a simple task. In a first step, an approach on this level may filter `.java` files by scanning the import statements for packages known to contain HTTP clients (e. g., `"import okhttp3."`). Subsequently, the approach may scan the candidate files for occurrences of call methods that relate to the identified HTTP client (e. g., `"execute("` or `"call("`). By including a trailing dot for imports and an opening parenthesis for call methods, the search does not find these strings in other contexts (e. g., `"call"` could also find a local variable `callerId`. Similarly, the approach may add further indicators and combine them for an overall score Gerdes et al. (2018).

This method can achieve a good recall, but cannot guarantee good precision. This over-approximation of calls happens due to very limited usable information. In particular, method names are not limited to a single class, but can appear multiple times (e. g., method `"execute"` in class `java.util.concurrent.Executor`).

Furthermore, clients with the call pattern *API Interface* do not necessarily use standard names, but the method calls are named individually by the client developers. This renders the approach infeasible for such clients.

This approach may extract call targets if they are hard-coded with the call and follow a predetermined template. Otherwise, call target extraction is not possible.

Overall, this approach may support manual analysis of HTTP communication or prepare a project for another detection method, but does not allow precise identification on its own.

CodeQL CodeQL (Avgustinov et al., 2016) is a framework to enable SQL-like queries for a code base. Originally intended for location of potential vulnerabilities by security analysis, it also allows general queries for code based on several queryable elements. Listing 5.16 shows a query for the OkHttp client.

CodeQL

```
1 from MethodAccess call, Method method
2 where
3     call.getMethod() = method
4     and
5     method.hasName("execute")
6     and
7     method.getDeclaringType().
8         hasQualifiedName("okhttp3", "Call")
9 select call
```

Listing 5.16.: A CodeQL Query to Identify HTTP calls with OkHttp.

The query searches for a method (line 3) called "execute" (line 5) provided by the class "Call" in the package "okhttp3".

This approach shares the general advantages and disadvantages with the direct text search. The additional query language to define the search does not influence the challenges of the text search approach. In particular, methods with common names (e. g., `execute`) may still be confused.

CodeDetectors Genfer and Zdun (2021) have introduced the concept of *code detectors* for the analysis of APIs in microservice architectures. Such code detectors are another form of advanced text search as they do not compute or use additional information. Instead, they rely on predefined template patterns that are matched against the source code files. Subsequently, a second phase compares the contents of identified "hotspots" with known API endpoints for the known microservices. Such links and the resulting graph structures are then further analyzed.

An example is the detection of `@GET("/timeservice?placeid=placeid")` by using the template `@GET("*")`.

However, this approach is not suitable for general settings as the possible call targets (API endpoints) in the microservice architecture need to be known in advance. Furthermore, the call target needs to be defined within the hotspot which is only one of four options for call target definition.

AST Analysis The Java compiler includes a lexer and a parser. They match elements of the source code to elements of the programming language and put them in relation to each other. The results is an abstract syntax tree (AST). In addition to the information present in the source code itself, the AST features fully resolved names for each node (e. g., `org.apache.hc.client5.http:execute`). Listing 5.17 shows an XPath detection query to be used with the tool *PMD*.

XPath query for the PMD tool

```
1 //Name[(//ImportDeclaration/Name[contains(@Image,
2 "org.apache.hc.client5.http")]
3     or //ImportDeclaration/Name[contains(@Image,
4     "org.apache.http")]) and ends-with(@Image, ".execute")]
5
6 //PrimarySuffix[(//ImportDeclaration/Name[contains(@Image,
7 "org.apache.hc.client5.http")]
8     or //ImportDeclaration/Name[contains(@Image,
9     "org.apache.http")]) and @Image="execute"]
```

Listing 5.17.: An XPath Rule for the Detection of the *Apache HttpClient*.

The query searches for a method with the exact name or a name containing "execute" (lines 4 and 9) somewhere in one of two subpackes for the Apache HttpClient.

Evaluation on sample projects shows better recall than the pure text search (Nitz, 2022). However, such general rules are still only applicable for some call patterns, but not for those with names that are individually set by the client developers.

Furthermore, the approach yields multiple results for some cases. In particular, for the HTTPUrlConnection, AST-based detection finds multiple method invocations that may trigger a call. However, the HTTPUrlConnection handles these internally to only perform the call for the first of these method invocations and stores the results for subsequent ones. These order of executions cannot be considered in the XPath rules.

Bytecode Analysis The Java compiler transforms the source code into byte code. This is the final intermediate form before the bytecode is executed on a Java Virtual Machine (JVM). Thus, it is the step that offers more information for static analysis than all of the others.

In addition of pure transformation of individual elements, modern Java compilers optimize byte code in a greater context. In particular, they may inject a call target that has been defined elsewhere directly into the respective method invocation. Listing 5.18 shows Java bytecode for an HTTP call.

Disassembled Java bytecode

```
48: astore_2
49: aload_1
50: aload_2
51: invokestatic #72 //Method
    java/net/http/HttpResponse$BodyHandlers.ofString:()
    Ljava/net/http/HttpResponse$BodyHandler;
54: invokevirtual #122 //Method
    java/net/http/HttpClient.send:(Ljava/net/http/HttpRequest;
    Ljava/net/http/HttpResponse$BodyHandler;)Ljava/net/http/HttpResponse;
57: astore_3
58: ret
```

Listing 5.18.: Java Bytecode for an HTTP Call Using the Java HttpClient.

Instruction 54 (lines 7–9) calls the method "send" in the class "HttpClient" with two arguments and records the result in a 'HttpResponse".

This byte code segment is disassembled from a .class file using the `javap` tool.

Evaluation on sample projects shows that this approach is slower than an analysis on AST-level, but the results are good with respect to precision and recall.

However, the recovery of call targets was only possible in a few cases. In particular, also this approach can never identify call targets that are not present in the project files statically.

AI-based Analysis Another distinct detection method is given by methods based on artificial intelligence (AI), in particular large language models (LLMs). These approaches utilize heuristics in a non-deterministic way. Since LLMs do not always provide reliable information, and hallucination cannot be ruled out, we decide to not further include such approaches in the current discussion.

Additional Analysis for Call Targets If an HTTP call is identified, but the call target is unknown, an additional dataflow analysis may be conducted. Such an analysis tracks the possible preceding operations on a String object to give possible values it can attend at the current position in the source code. However, such approaches show a large variation in precisions, e. g., Li et al. (2015) report for the same tool precision values between 7 % and 70 % depending on the precise task and evaluated project.

This approach does not pose an alternative to the previously presented options, but an additional component for a extensive detection. If the call target information is not provided statically in the project's files, now additional information can be retrieved.

5.3 Conclusions

Although HTTP calls take a large variety of forms in Java, the static detection of such calls is possible. Several detection options present trade-offs between effectiveness and efficiency. In general, the precision of such approaches tends to be low, while the recall is good. Thus, these analysis would serve well as a pre-step for an extending analysis, i. e., by identifying hotspots for further inspection.

On the other side, the insights form the empirical analysis show clearly, that call targets are not always provided in the project's files. Thus, a static analysis cannot be able to identify such call targets in general systems reliably.

Chapter 7 will consider this discussion for an overall evaluation of static approaches for detecting usage of deprecated remote APIs.

Static Analysis of Remote APIs

6

Complementing the discussion of API clients in the previous chapter, this chapter discusses statically available information on API providers.

Developers describe local APIs in terms of the programming language or additional tools of the ecosystem (e. g., Javadoc). Since providing and using the information happens within the same ecosystem, the documentation by the provider is visible and understandable to the client. This may include method signatures Annotations, and Javadoc comments in Java.

In contrast, HTTP APIs may be implemented in different languages. In particular, client and provider of an API do not need use the same programming language. Thus, there is in general no API documentation in the providers' code that the client developer could see and understand. Therefore, if HTTP APIs are documented for clients, such documentation is external to the API and its underlying software system.

We can now add the third research sub-question for RQ 1:

RQ 1 **How can existing static analysis techniques for local APIs be adapted to detect usage of deprecated remote APIs?**

RQ 1.3 How is deprecation of HTTP APIs documented in statically available sources?

We distinguish dedicated API description formats (e. g., API blueprint, Swagger, and OpenAPI) and other less formal sources of information (e. g., release notes, or blog posts). For the description formats we focus the current de-facto standard *OpenAPI* (Ponelat and Rosenstock, 2022; Serbout and Pautasso, 2024a).

Publication Note The contents of this chapter have not been published individually yet. Parts of Section 6.1 have been developed within the master's thesis (Hoffmann, 2026)[1].

6.1 Deprecation Information in OpenAPI Descriptions

OpenAPI descriptions offer information on an API in a standardized format readable for both, humans and machines. Listing 6.1 shows an excerpt from an OpenAPI description for the timeservice example with added deprecation information.

[1] This thesis is not publicly available. The relevant contents are reproduced here with the author's consent

OpenAPI (YAML)

```yaml
openapi: 3.0.3
info:
    title: Time Service API
    version: 2.4.0
paths:
    /timeservice:
        get:
            summary: Get current time for a place
            description: Returns the current date and time for a provided location. The parameter needs to be one of the predefined IDs provided by /places/. The returned value is a string in ISO 8601 format. This operation is deprecated in favor of /place/{placeId}/time and will be removed in version 3.0.0.
            deprecated: true
            parameters:
                - name: placeId
                  in: query
                  required: true
                  schema:
                      type: string
```

Listing 6.1.: An OpenAPI Description for the Timeservice API with Deprecation Information.

The deprecation information is documented in natural language in the description field (line 9). Furthermore the deprecation tag is set (line 10). Both ways of documentation are optional.

In contrast to static APIs, the actual call target and its documentation are not necessarily linked to each other. Thus, discovery of a corresponding OpenAPI description is an additional task. We will discus further this in Section 6.2.

To deprecate such an entire API, a single operation or another element multiple options exist. An unambiguous way is the boolean flag `deprecated` that can be set to true or false. However, since it is an optional tag, it may be missing. In this case, no interpretation of the deprecation status in possible.

6.1.1 Research Design

Research Objective Following the call and target identification, we now focus the second essential step in deprecation detection: checking the called API's deprecation status. For this goal, this chapter investigates RQ 1.3

(How is deprecation of HTTP APIs documented in statically available sources?) for the special, but widespread case of OpenAPI descriptions.

Related Work OpenAPI descriptions are often target of research on HTTP APIs. Serbout and Pautasso (2024a) constructed and shared a large dataset of 1,275,568 publicly available OpenAPI description documents. Di Lauro et al. (2022) and Serbout and Pautasso (2024b) studied evolutionary and versioning practices across multiple versions of the same API.

A focus of deprecation is added by Yasmin et al. (2020). They built *RADA*, an "RESTful API Deprecation Analyzer" that checks an OpenAPI description for deprecation in two ways: the deprecation tag, and the occurrence of keywords like "deprecated" in descriptive fields. However, due to the pure keyword-based search without additional filtering, false positives occur (e. g., "This operation replaces the **deprecated** operation ..."). Furthermore, the study's goal was not the characterization of the current usage of deprecation in OpenAPI descriptions. Thus, the empirical evaluation was performed on the smaller *apis.guru* dataset with 1,068 OpenAPI descriptions of which 219 were used. In addition, the publication does not report how many deprecated elements feature deprecation tags or deprecation information in a textual description.

Research Method We base our research approach on the study design by Yasmin et al. (2020), but analyze the APIstic dataset by Serbout and Pautasso (2024a). For this, we obtained an updated dataset from the authors in April 2025.

Similar to Yasmin et al., we start with a text search for keywords like "deprecated" on a small randomly sampled subset of OpenAPI descriptions. We then manually classify the results: for actually deprecated operations, we are interested in the following features:

- **Textual Documentation:** Does the OpenAPI description provide some textual documentation?
- **Textual Deprecation:** Does a textual documentation mention deprecation?
- **Deprecation Tag:** Is the deprecation tag set to `true`?
- **Alternative:** Is an alternative to the deprecated operation or similar advice for client developers mentioned?
- **Reason:** Does a textual documentation explain a reason for deprecation?
- **Time of Deprecation:** Does the textual documentation mention when the deprecation was introduced?

- **Sunset:** Is indicated if the operation will be removed in the future?

Meanwhile, we exclude OpenAPI descriptions if their textual information is not provided in English, or if the operations found to be not deprecated (e. g., the deprecation tag is set to false). Furthermore, we excluded near-duplicates, i. e., we only added the result once per OpenAPI description if the same deprecation information was provided for multiple operations (e. g., multiple HTTP methods for the same path). After exclusion and de-duplication, 216 OpenAPI descriptions remain – a very similar number to the analysis by Yasmin et al.

Since a manual inspection of more than one million OpenAPI descriptions is not feasible, we are interested if the small subset is representative for the entirety it is sampled from. Therefore we design three machine learning approaches using a Naive Bayes classification algorithm, a support vector machine algorithm, and fine-tuning the large language model BERT. While the first two approaches fail to identify deprecated elements altogether with precision and recall below 80 %, the BERT-based approach yields accuracy, precision and accuracy values of at least 95 %. Note that it is not a goal to apply such classification as part of the deprecation detection framework, but only to generalize our insights on the practice on deprecation from the small manually analyzed subset to a greater population. Thus, we do not introduce and discuss these approaches in detail here, but focus on the insights gained as they are already the result of the manual inspection. Chapter 13 will include a discussion on the possible inclusion of machine learning and artificial intelligence methods for the deprecation framework.

6.1.2 Results

Documentation of Deprecation While 96 % of deprecated API elements feature a textual documentation in a summary or description field, only 62 % of deprecated API elements include the deprecation in this textual documentation. Similarly, we found 55 % of deprecated API elements to feature a deprecation tag set to `true`. The overlap of the two ways to indicate deprecation is small with only 17 % of deprecated API elements.

In a few instances, deprecation was signaled through a custom tag, e. g., `x-deprecation` or `x-deprecated`. These were rarely used instead of or in addition to the standard `deprecated` tag and only occurred for deprecation of parameters or schemata. These element only feature a standard `deprecated` in OpenAPI version 3, but not in previous versions. Thus, the need for an extension by using an "`x-`" tag arose.

Documentation of Additional Information While 37 % of deprecated API elements provide an alternative for client developers to use instead, information on reasons for the deprecation (12 %), the time of deprecation (7 %), or sunsetting (5 %) are rarely found.

If a reason for deprecation is provided, we found those to often directly link to an alternative, e. g., renaming or moving operations to another path, or deprecating an operation affected by an security issue in favor of a more secure alternative. We only found two reasons that do not feature a better alternative: One API was deprecated for removal, because the generated load on the database was too high to handle. Another API was deprecated, but not intended for removal, because the API providers found the documentation of this feature to be in bad state and thus, advised against using the API until the documentation would improve.

Yasmin et al. (2020) report similar numbers only on the level of OpenAPI descriptions (i. e., documents or files), but not for individual API elements. However, they similarly found that only in a minority of cases alternatives are reported, and other information is lacking almost always.

6.2 Discovery of OpenAPI Descriptions

Even if an OpenAPI description for an endpoint exists, there is no standard for its location. This contrasts the situation from local APIs where the signature of a method would immediately reveal its deprecation to the compiler.

Instead, the API provider is free to locate the OpenAPI description for an API at an arbitrary location and then link there from, e. g., an informational website. Common locations include a `docs.` or `dev.` subdomain and a `/docs` path. Common filenames include generic options for OpenAPI and alternatives specific to the API they describe.

To solve these inconsistencies, the RFC 8615 Nottingham, 2019 proposes a solution by introducing the concept of “Well-Known URIs” that should be provided at a uniform location. For API descriptions, this URI should be:

```
/.well-known/api-catalog
```

A server may redirect this URI to the actual location of the file.

Table 6.1 shows examples for common locations.

However, there is no guarantee that one of these locations is used. We also found OpenAPI descriptions in completely different locations. An example based on a real API, but transferred for the time service example is given by:

```
static.time.example/documents/timeservice/swagger-v3.yaml
```

Note that this example also uses OpenAPI's predecessor's name "Swagger" although it is an OpenAPI description in OpenAPI version 3 and the last Swagger version was 2.0.

Tab. 6.1.: Common Patterns for Location and Path to OpenAPI Documents

Approach	URL
Common Subdomain	docs.time.example/openapi.yaml
	docs.time.example/timeservice.yaml
	dev.time.example/openapi.yaml
	developer.time.example/timeservice.yaml
Common Path	time.example/docs/openapi.yaml
	time.example/docs/timeservice.yaml
Well-Known URI	time.example/.well-known/api-catalog

The lack of explicit association also affects the other direction: If an OpenAPI description is found, the provision of a server URL is optional and thus, it is not possible in these cases to verify that the correct OpenAPI description has been found. Listing 6.2 shows the server tag for the explicit provision of a corresponding server URL.

OpenAPI (YAML)

```yaml
openapi: 3.0.3
info:
    title: Time Service API
    version: 2.4.0
servers:
    -   url: https://time.example
```

Listing 6.2.: OpenAPI Description for the Timeservice API with Server URL.

The OpenAPI description features a tag "url" that explicitly mentions the location of the API. This allows the reliable mapping between API and its Implementation

6.3 Deprecation Information in Less Formal Resources

In addition or as an alternative to formal deprecation in an OpenAPI description, an API element may also be deprecated by other, less formal ways of communication.

A common way to provide deprecation information less formally is through changelogs. API providers may publish these with each new version of the API and client developers are often aware of such changelogs. We found an example for deprecation in the changelog of the Hetzner cloud services[2]:

> "The response field `deprecated` of all endpoints that return "ISOs" is deprecated. Please use the new field `deprecation` instead."

Other ways of informal communication comprise websites, blogs and newsletters. An example is given by Dropbox announcing the deprecation and sunsetting of their entire API in version v1 trough a post "API v1 is now deprecated" to their official blog[3].

6.4 Conclusions

We found deprecation information provided in OpenAPI documents to be well suited source – in particular, if the deprecation tag is used. If such documentation is provided, additional information featured (e. g., regarding alternatives) may be extracted and presented to the client developer.

However, multiple challenges remain: The discovery of OpenAPI documents is an non-trivial, but required step. If deprecation information is missing in OpenAPI documents, it cannot be assumed that the element is not deprecated. Deprecation information in textual descriptions requires advanced text parsing.

We are now able to answer research sub-question RQ 1.3:

Answer to RQ 1.3 *How is deprecation of HTTP APIs documented in statically available sources?*

The OpenAPI specification provides a format to formally signal deprecation of an API element. We found that real-world OpenAPI documents only partly use this feature, but instead or additionally use textual descriptions. If deprecation is documented in the OpenAPI description, the deprecation

[2] docs.hetzner.cloud/changelog, published 2023-10-12, visited 2026-05-10
[3] dropbox.tech/developers/api-v1-deprecated, published 2016-06-28, visited 2026-05-10

status is unambiguous, but if deprecation is neither acknowledged or declined, the deprecation status remains unknown.

Furthermore, deprecation may also be documented in less formal documents. All of these documents, formal and informal, share challenges in discoverability.

7 Discussion of Static Approaches

Following the separate analyses of clients and providers, we now join the views for a general impression of static approaches for the detection of usage of deprecated APIs.

7.1 Advantages and Disadvantages of Static Approaches

Static analysis examines a program without executing it. Thus, it can reason "over all possible behaviors that might arise at run time" (Ernst, 2003). This means, that a call does not have to be executed within an observation to be part of the analysis.

However, there are several disadvantages to static analyses: First, they are usually applied in a conservative manner, i. e., reporting results only if they are unambiguously true. Second, they can only use information available at the time of analysis, in particular, no exemplary runtime information is included.

The information available at runtime might include important properties of the environment collected at that time. Contrarily, a static approach can not foresee possible changes to the environment before the program is actually executed. In the setting of remote API deprecation, the deprecation may be issued after the static analysis.

7.2 Assessment of Static Approaches for Deprecation Detection

From local approaches we have generalized a three-step process: First, the call and its target are detected. Second, the target is checked for its deprecation status. Third, deprecation is reported if found.

For the first step, we found the identification of calls to be possible in some cases, but without guarantees. However, the identification of call targets is severely limited.

For the second step, we found OpenAPI descriptions to be an easily usable source of information. However now all API providers use these or provide the necessary information.

For the third step, the approach of static or dynamic engagement is not relevant. Even though we have not discussed this step yet, some options are evident, e. g., if usage of a deprecated API is found, the detector writes an issue to a ticket system.

In general, the three-step process derived from local APIs is also applicable to the static detection of usage of deprecated remote APIs. Such an approach might yield high precision, but low recall is expected due to missed calls or call targets in the first step and missed or unknown deprecation information in the second step.

However, a core problem arises in the dynamic nature of remote calls: While as local API is built into the same execution package by a compiler, a remote API needs to be called at runtime. Thus, deprecation detection at one point in time (e. g., when writing the code or building an executable) does not guarantee continued correctness for the program. A later call to a deprecation API should also be reported to the client developer to react accordingly. This would require the repeated execution of the static analysis framework, e. g., once a day or once a month depending on the importance of the API call.

In summary, we are now able to answer research question RQ 1:

Answer to RQ 1 *How can existing static analysis techniques for local APIs be adapted to detect usage of deprecated remote APIs?*

For all essential steps in the generalized deprecation detection process a static analysis technique for remote APIs is available. However, their effectiveness is clearly limited due to the nature of remote APIs using dynamically determined targets and also a possibly dynamically changing deprecation status.

Part III

Dynamic Deprecation Analysis

Dynamic Analysis of HTTP Clients in Java

8

We now tend towards dynamic approaches. For this, we will be able to reuse insights we gained and described in Part II for the static analysis. We complement this foundational information with concrete options and approaches using dynamic analysis in this chapter and the subsequent Chapters 9 and 10.

Therefore, we discuss research question RQ 2 and introduce the first research sub-question RQ 2.1:

> **RQ 2** **Which runtime properties of remote API calls are relevant to identify usage of deprecated remote APIs?**
>
> RQ 2.1 Which runtime information is relevant and available for API clients and their requests?

To answer this research question and its sub-questions, we follow an ad-hoc approach combining gray literature, and engineering of prototypical tools.

Note that in dynamic analyses we cannot only study the client and the provider, but also the messages sent between them. We subsume request messages as part of the client's communication (this Chapter 8) and response messages as part of the provider's communication (the subsequent Chapter 9).

Publication Note The contents of this chapter are partly presented in (Bonorden and van Hoorn, 2024a). Additional insights were gained through the Bachelor's thesis (Müller, 2024).

8.1 Observability

Observability is the capability of a distributed software system to recreate execution traces by recording and combining individual measurements (Majors et al., 2022). Hausenblas adds two aspects in his definition – the need for a continuous analysis and the goal of feedback by using the observations to re-influence the system:

> OBSERVABILITY
>
> *Observability* is the capability to continuously generate and discover actionable insights based on signals from the system under observation, with the goal of influencing the system. (Hausenblas, 2024)

Observation requires instrumentation of client code, i. e., specifying for each runtime component which information is to be provided to the observation framework. Such information is called “telemetry” and may take the form of logs, metrics, or traces. Observation frameworks are provided by commercial vendors or as research software – e. g., the Kieker Observability Framework (Yang et al., 2025). The Cloud Native Computing Foundation provides *OpenTelemetry* as an open standard format for telemetry data together with foundational tools (Young and Parker, 2024).

8.1.1 OpenTelemetry

OpenTelemetry defines formats for the telemetry data itself, i. e., logs (textual event descriptions), metrics (measurements of properties), and traces (executions of a request possibly through multiple components, usually split into multiple spans). Furthermore, a format for recorded meta-data (“baggage”) is provided.

In addition to the format definitions, the OpenTelemetry framework provides tools for fundamental tasks in the context of observation. For the goal of detection of deprecation, the following two tools are of special interest:

- An instrumentation framework adds collecting of relevant information to executable code.
- An *OpenTelemetry Collector* processes raw telemetry data and exports the results.

We will utilize these in Section 8.2.

8.1.2 Relevant Information in HTTP Requests

The format of HTTP requests (and responses) is defined through various RFC documents. In this subsection we present the properties that are relevant to deprecation detection. An example for a HTTP request is given in Listing 8.1.

HTTP message

```
1 GET /timeservice?placeId=germany/hamburg HTTP/1.1
2 Host: time.example
3 Accept: text/plain
```

Listing 8.1.: A Header in a HTTP Request for the Timeservice Example.

While other HTTP operations may have a body, it is usually empty for GET.

In line 1, the header indicates the protocol (HTTP in version 1.1), the HTTP method used (GET) and a partial URL for the call. In line 2, the host part of the URL is added. In line 3, the request provides one or multiple preferred formats for the response's body.

Each HTTP request contains information about the message's target, split into the host (`time.example`, i. e., the domain to be resolved for an IP address) and the URI (i. e., the remainder of the URL) (Fielding et al., 2022b). The URI may contain the path to the resource (`/timeservice`), a query (`?placeId=germany/hamburg`), or a fragment (starting with `#`, not present in this example).

Furthermore, the request provides the protocol (HTTP) its version (e. g., 1.1) and the HTTP method (e. g., GET)

For some HTTP methods (e. g., POST), a request body can be added. The body is transmitted in the same message as the header – separated by an empty line.

For the goal of deprecation detection, we are mostly interested in the path (host and URI) to the requested API. This aligns with the first step of the generalized three-step model of deprecation detection: By observing the HTTP request, we know there is a call, and by reading host and URI, we extract the call target.

We will similarly discuss relevant information in HTTP responses in Section 9.1.

8.2 Runtime Data Collection with OpenTelemetry

Instrumentation The OpenTelemetry framework offers three ways of instrumentation. When using *manual instrumentation*, developers import a library and manually declare which information to export. For some cases, e. g., the Apache HttpClient, an *instrumented library version* is offered that integrates the instrumentation code into the library itself. Finally, an *auto-instrumentation* feature may analyze the bytecode of a Java application and injects observation commands for known communication patterns – in

particular, for HTTP clients. This analysis is similar to the bytecode analysis we provided in Section 5.2.

Using the auto-instrumentation, we can modify which request header information should be recorded. The start line of an HTTP request is always tracked and the host field is also part of the standard execution. Nevertheless, Listing 8.2 shows the explicit configuration.

OpenTelemetry Instrumentation Configuration (System Property)

```
1 otel.instrumentation.http.client.capture-request-headers=
2     "host"
```

Listing 8.2.: Configuration of the Auto-instrumentation for OpenTelemetry Regarding Requests.

In this example, only one additional header ("host") shall be recorded. All other relevant headers, e. g., the URL, are recorded automatically.

Most of the HTTP clients identified in Chapter 5 are available for auto-instrumentation. Some clients are not directly supported by OpenTelemetry, but use another HTTP client internally, that is subject to the auto-instrumentation. One notable exception is the *Volley* HTTP client: Since Volley has not been maintained since August 2021, OpenTelemetry decided to also suspend the auto-instrumentation for it. However, we found that the client is still used in recently modified repositories.

Collector In addition to the instrumentation, we need to configure the collector. For a proof-of-concept we do not need additional features like data filtering or other pre-processing, but are only interested in exporting all the data from OpenTelemetry. For this a simple file exporter to a JSON file as shown in Listing 8.3 suffices.

OpenTelemetry Collector Configuration (YAML)

```
1 exporters:
2     file:
3         path: ./output.json
4 service:
5     pipelines:
6         traces:
7             exporters: [file]
```

Listing 8.3.: A Configuration for the OpenTelemetry Collector.

Lines 1–3 configure the path for the file exporter. Lines 4–7 instruct the collector to use this file exporter for all traces.

This configuration yields a file containing recorded spans (part of a trace). For each span, the recorded attributes are provided as shown in Listing 8.4.

JSON

```
"attributes":[
    {
        "key":"url.domain",
        "value":{"stringValue":"time.example"}
    },
    {
        "key":"url.path",
        "value":{"stringValue":"/timeservice"}
    },
    {
        "key":"url.query",
        "value":{"stringValue":"?placeId=germany/hamburg"}
    },
    {
        "key":"url.full",
        "value":{"stringValue":
        ↪  "https://time.example/timeservice?placeId=germany/hamburg"}
    },
    {
        "key":"http.request.method",
        "value":{"stringValue":"GET"}
    }
]
```

Listing 8.4.: A Partial Span as Recorded and Exported by OpenTelemetry.

The recorded HTTP attributes show fundamental information about this request, i. e., the HTTP method and the call target's URL.

8.3 Runtime Data Collection without Additional Frameworks

While the auto-instrumentation generates low overhead locally (Reichelt et al., 2024), such individual instrumentations can accumulate to significant performance issues (Hammad et al., 2025). Furthermore, complexity is added to the development project, if an observability framework is newly introduced only to cover deprecation detection.

As an alternative we have tested a manual implementation of the recording and export functionality as an extension to an Java HTTP client. Note that

this is not manual instrumentation in the sense of OpenTelemetry instrumentation as it does not target any observability framework, but processes the information locally. For this, we extended the Apache HttpClient and the OkHttp client by using interceptors – a concept both libraries provide for extensibility.

Here, we do not follow this approach further, but note that the idea of extracting runtime information data is not bound to observability frameworks. Instead, it can be conducted locally within Java HTTP clients. The complete presentation of this option is available in (Müller, 2024).

8.4 Conclusions

We briefly discussed observability and telemetry. These concepts and tools allow us to record information about HTTP requests on the client' side. Through manual configuration a variety of information is available and exportable. This allows us to anser the research sub-question RQ 2.1:

Answer to RQ 2.1 *Which runtime information is relevant and available for API clients and their requests?*

Collecting runtime data relies on the detection of HTTP communication at runtime. This is possible through an additional observability framework or within the HTTP clients themselves. Subsequently, the entire request with its meta-information is available.

The most important property of an HTTP request for the purpose of deprecation detection is the call target. Such a call target is recorded by the auto-instrumentation of OpenTelemetry, but could also be exported without an observability framework. The information about the call target may be given in a full URL or in slices, e. g., host and path.

Dynamic Analysis of Remote APIs

9

Adding the provider and response perspective, this chapter complements the previous one by adding the second sub-question to research question RQ 2:

RQ 2 **Which runtime properties of remote API calls are relevant to identify usage of deprecated remote APIs?**

RQ 2.2 Which runtime information is relevant and available for API providers and their responses?

Publication Note The contents of this chapter are partly presented in (Bonorden and van Hoorn, 2024a).

9.1 Deprecation in HTTP Response Metadata

Following Chapter 8, we continue the discussion in the context of the OpenTelemetry instrumentation and OpenTelemetry collector. Similar to that discussion, Listing 9.1 presents a HTTP message – this time, a HTTP response.

HTTP message

```
1 HTTP/1.1 200 OK
2 Content-Type: text/plain
3 Deprecation: @1688169599
4 Sunset: Thu, 31 Dec 2026 23:59:59 GMT
5
6 2026-07-21T10:00:00Z
```

Listing 9.1.: Header and Body in an HTTP Response for the Timeservice Example. Header and body are separated by the first empty line.

In line 1, the header provides the protocol and protocol version, and a status code in numerical (200) and textual (OK) representation. In line 2, the header adds metadata for the format of the body.

The lines 3 and 4 contain deprecation related header fields, `Deprecation` and *Sunset*. Both fields represent date and time, but use different formats for this information.

The body contains date and time in a third format.

Each HTTP response contains information about the message's protocol (HTTP), its version (e. g., 1.1) and the status for the corresponding request: The status is given as a numerical code (e. g., 200 for a successful request) and a textual representation (e. g., `OK` for a successful request).

In contrast to a GET request, the response usually contains a body. For this, a header field *Content-Type* indicates the format and information in this format is provided after the header fields – separated by an empty line.

In addition to the content type, more header fields are represented as name–value pairs. The header fields related to deprecation detection will be discussed in the following.

Relevant Information in HTTP Responses The `Deprecation` header field is designed to document of the called API element (Dalal and Wilde, 2025). Its value is a date and time element given Unix time, i. e., in seconds from 1970-01-01 at midnight. A leading "@" is used to distinguish this date and time format from arbitrary integers. An example is given by

```
Deprecation: @1688169599
```

indicating that the deprecation was introduced on 2023-06-03 at 23:59:59.

The `Sunset` header field provides a date for the API element's removal (Wilde, 2019b). The date is given in "HTTP date format" and an example is

```
Sunset: Thu, 31 Dec 2026 23:59:59 GMT
```

for the removal of this element on or after the end of the year 2026.

In addition to these directly relevant fields, also `Link` fields could be of interest (Wilde, 2019a). These are used for multiple types of relations. In particular, a "service-doc" link may point to human readable API documentation, and a "service-desc" link presents a link to formal API description, e. g., an OpenAPI document. Examples are

```
Link: "<https://time.example/api>; rel='service-doc'"
```

and

```
Link: "<https://docs.time.example/>; rel='service-desc'"
```

for service-doc and service-desc, respectively. It is common that multiple link header fields appear in one response.

Further fields may contain deprecation-related information, but are rarely used. Custom header fields had to be used prior to the definition of the standard `Deprecation` header. At that time, such custom fields were supposed to start with "`X-`", e. g., `X-API-Deprecation-Info` (Saint-Andre et al., 2012). Another alternative was using the general `Warning` header field (Fielding et al., 2014) that itself has been deprecated by 2022 (Fielding et al., 2022a).

9.2 Collecting Deprecation Information from HTTP Responses

For the collection of HTTP response data multiple options arise. Although, we have associated the response with the API provider that generated it, we can also record it once it arrives back at the client. Furthermore, we could theoretically intercept the response message itself during the communication.

Collection with OpenTelemetry The first option mirrors the data collection for the request. Therefore, we use auto-instrumentation to instruct the HTTP clients in Java to record desired information. Listing 9.2 shows the configuration to capture the most important header fields for deprecation detection as discussed above.

OpenTelemetry Instrumentation Configuration (System Property)

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

Listing 9.2.: A Configuration of the Auto-instrumentation for OpenTelemetry Regarding Responses.

This configuration instructs to record three additional headers for responses. Fundamental information (e. g., the status code) is recorded automatically without explicit configuration.

In addition, we again need to instruct the OpenTelemetry Collector to export the collected data. Since we still don't require any filtering or preprocessing, we can reuse the configuration presented in Section 8.2 as it already includes the export for all data fields recorded.

Again, we then yield records of spans with HTTP attributes. We the configuration from Listing 9.2, they now contain information from the response header fields. Listing 9.3 shows an example.

JSON

```
"attributes":[
    {
        "key":"http.response.header.deprecation",
        "value":{"arrayValue":{"values":[
            {"stringValue":"@1688169599"}
        ]}}
    },
    {
        "key":"http.response.header.sunset",
        "value":{"arrayValue":{"values":[
            {"stringValue":"Tue, 31 Dec 2024 23:59:59 GMT"}
        ]}}
    },
    {
        "key":"http.response.header.link",
        "value":{"arrayValue":{"values":[
            {"stringValue":"<https://docs.time.example/>;
            ↪  rel=\"service-desc\""}
        ]}}
    },
]
```

Listing 9.3.: A Partial Span as Recorded and Exported by OpenTelemetry.

This span includes a deprecation field with a respective date in the Unix time format and a sunset field with a respective date in HTTP date format. Furthermore, a link to a machine-readable API description is provided.

These headers are recorded in arrays, since each field could appear more than once.

Collection in the Client without OpenTelemetry Müller (2024) did not only record and analyze request information as described in Section 8.3, but also the response messages. We cover the process briefly for the Apache HttpClient, but mainly note again, that collecting runtime data is not bound to an observability framework.

The Apache HttpClient defines an Interface `HttpResponseInterceptor` that a custom class may implement. The interface defines a method `process` that is called when a client with such an interceptor receives an HTTP response. Listing 9.4 shows the essential parts of the class. To build such a client the Builder provided by the library offers a dedicated `addResponseInterceptorLast(interceptor)` method.

```java
final Set<String> deprecationHeader = new HashSet<String>();
deprecationHeader.add("deprecation");
deprecationHeader.add("sunset");

@Override
public void process(final HttpResponse httpResponse, final EntityDetails
↪ entityDetails, final HttpContext httpContext) throws HttpException,
↪ IOException {

    final boolean deprecated =
    ↪ Arrays.stream(httpResponse.getHeaders()).anyMatch(header ->
    ↪ deprecationHeader.contains(header.getName()));
    httpContext.setAttribute("deprecated", deprecated);
}
```

Listing 9.4.: Deprecation Detection in the Class `DeprecationInterceptor` as an `HttpResponseInterceptor` for the Apache HttpClient.

The class uses a set *deprecationHeader* to define the headers indicating deprecation. If a response is received by the Apache HttpClient, the process method defined here is then called. This simple implementation traverses the response headers to find indication for deprecation. Similarly, other reactions or an export would be possible.

The last line sets an externally readable attribute for further interaction.

Collection outside of the Client Finally, in the dynamic remote setting, recording request and response data is not strictly bound to the client and provider of the API. Instead, network traffic can be observed between the two communication partners, e. g., with a tool like Wireshark[1]. However, this approach comes with two essential limitations: First, the network traffic needs to be observable, i. e., the tool needs to be executed in the right computer network. Second, the encrypted methods of the commonly used HTTPS protocol need to be decrypted, which would require to either work together with the client application to use the actual decryption keys or to sabotage the encryption altogether. Neither option is suitable for the goal of deprecation detection as recording in the client is better in effectiveness and efficiency.

9.3 Conclusions

The data collection for the client and its requests can be extended to record also the provider's communication in the response message. HTTP response header fields like *Deprecation* or *Sunset* carry exactly the needed information if they are used. However, since the final version of the Deprecation standard has only been proposed in March 2025, and the Sunset header field is not a standard, but only an informational suggestion, no widespread use of these headers could be observed yet. This gives the answer to research sub-question RQ 2.2:

Answer to RQ 2.2 *Which runtime information is relevant and available for API providers and their responses?*

Similarly to HTTP request data, collecting runtime data of HTTP responses relies on the detection of HTTP communication. Again, an observational framework may be used or the task is assigned to the HTTP clients themselves.

The most important HTTP header fields for the deprecation detection are *Depreaction* and *Sunset*. For additional analysis a *Link* header may point to an OpenAPI description of the API.

[1] wireshark.org, accessed 2026-05-02

10 Discussion of Dynamic Approaches

Following the separate analyses of clients (together with their requests) and providers (together with their responses), we now join the views for a general impression of dynamic approaches for the detection of usage of deprecated APIs.

10.1 Advantages and Disadvantages of Dynamic Approaches

Dynamic analysis executes a program and observes such executions. Thus, it can work with actual data and "is precise because no approximation or abstraction need be done" (Ernst, 2003). This means that all desired events and metrics can be recorded or calculated.

However, also the dynamic analysis features several disadvantages: First, information obtained from an observed execution cannot be generalized to other executions as data may change and in general no connection is derivable from the dynamic analysis alone. Second, special situations may be over-represented or under-represented in the set of observed executions. An example for over-representation is a recurring, but rather unimportant HTTP call to an autocompletion API for an input task. An example for an under-representation is a rarely used, but important HTTP call to a time synchronization API on the first day of a year.

In Chapter 7 we have noted that a static approach only records the environment at the time of its execution, i. e., the remote API's deprecation status. A dynamic approach can, on the contrary, determine the deprecation status at the time of the actual call to that API. This implies, that a dynamic approach has to be executed either within the real execution (possibly on client devices) or continuously with good test cases in parallel to real executions.

10.2 Assessment of Dynamic Approaches for Deprecation Detection of Remote APIs

We recall the generalized three-stop process for deprecation detection: First, the call target and its target are detected. Second, the target is checked for its deprecation status. Third, deprecation is reported if found.

For the first step, we found recording runtime information a feasible approach – with or without a deprecation framework. If the detection utilizes auto-instrumentation, it's correctness and completeness depends on the auto-instrumentation capabilities for the exact client used. In contrast to

static analysis, the identification of call targets is simple in the dynamic analysis as they are explicitly given in every HTTP request.

For the second step, we found the same approach as for the first step to be useful. If deprecation information is provided in the response's metadata it can easily be extracted. However this communication is not often used.

For the third step, we did not discuss dynamic options yet. However, the same approaches as for static analysis are applicable, e. g., writing an issue to a ticket system.

Answer to RQ 2 *Which runtime properties of remote API calls are relevant to identify usage of deprecated remote APIs?*

For all essential steps in the generalized deprecation detection process a dynamic analysis technique for remote APIs is available. The main limitation for this approach arises in the rare usage of deprecation information in response meta-data. Another limitation is presented by depending on auto-instrumentation, but this is rather minor.

Part IV

Results and Discussion

11 Hybrid Approach

The preceding Parts II and III have analyzed options for static and dynamic analyses, covering clients and their requests as well as providers and their responses. Table 11.1 summarizes the options.

This chapter now proceeds to evaluate the insights gained to engineer a hybrid solution to combine advantages of both detection approaches while minimizing disadvantages. To this end, we will derive design principles, construct an abstract approach, and describe a prototypical implementation. The subsequent Chapters 12 and 13 will evaluate the implementation and discuss the results in a greater context, respectively.

11.1 Design Principles

The analysis of deprecation for local APIs, empirical studies for remote APIs, and prototype construction in Parts I to III yield the following design principles for the design of the envisioned solution.

DP 1 The client analysis should not be limited to a single programming language or ecosystem, or to a single communication client library.

DP 2 The client analysis should include call targets that are only available or determined at runtime.

DP 3 The deprecation analysis should include the latest available deprecation information from all relevant sources.

DP 4 The deprecation analysis should not require additional prior knowledge about the specific API.

Design Principle DP 1 The foundational discussion in Part I has shown that a widely applicable solution should be preferred to a collection of individual solutions per environment. Furthermore, within an environment like Java, all ways to represent an external call should be covered. We have seen with the dynamic analysis and its auto-instrumentation that such a goal is achievable.

Design Principle DP 2 According to our analysis in Part II, the static approach is mostly limited on the client side. On the contrary, the dynamic analysis in Part III shows good coverage for all types of call targets. Thus, we aim to achieve the scope of the dynamic analysis.

Tab. 11.1.: Overview of Static and Dynamic Analyses for Clients and Providers

	Client	**Provider**
Static detection	Source code analysis on text, AST or bytecode level with limited access to call targets.	Analysis of OpenAPI documents for deprecated tags and textual information with limited discoverability.
Dynamic detection	Auto-instrumentation of client code with simple detection of calls and their precise targets.	Auto-instrumentation of client code with detection of deprecation in meta-data but limited usage in practice.

Design Principle DP 3 Both, the static and dynamic analysis have shown disadvantages in performing the second step of the general deprecation detection process: checking the deprecation status of an API. With a static approach, the discovery of a suitable OpenAPI description is a persisting challenge. With a dynamic approach, only the rarely used deprecation header is cover, but not deprecation in an OpenAPI description which is used more often. Thus, we aim to combine all available sources of information. In addition, we need to use the latest version available for the detection as the deprecation status of an API can change independently of client evolution.

Design Principle DP 4 To keep the framework as generalizable as possible, we should not require any additional information about the API or its usage other than the actual code used and messages sent. However, this does not prevent from provision of *optional* additional information, e. g., manual supply of a link to an OpenAPI description.

11.2 System Overview

11.2.1 General process

We now combine static and dynamic approaches to engineer a hybrid approach. First, we provide an architectural overview that mimics the generalized deprecation detection process derived from local APIs. This architecture sketch is shown in Fig. 11.1.

For the first step, detecting calls and their targets, we choose the dynamic approach of data collection in the client – possibly through an observability framework like OpenTelemetry. Here, we can observe the request as well as the response, and export this data. This is the *observation*.

For the second step, checking the deprecation status, we combine all available static and dynamic analyses, i. e., meta-data in the HTTP response, OpenAPI descriptions, and possibly other sources. This is the *detection*.

For the third step, reporting the findings, we use an intermediate action by saving the findings to a file. From here, a variety of options arises to communicate the findings further. This is the *report*.

11.2.2 Observation

We instruct the client to record all relevant information from requests and responses. This includes the standard record of the call target from a request as well as custom instructions to record relevant header fields from

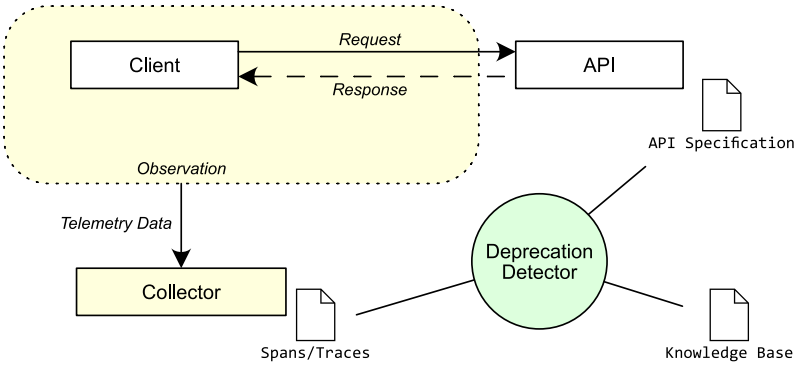

Fig. 11.1.: Structural Overview for the Deprecation Detection Framework.

The observed system is given as client and API together with their request–response communication. We observe the client-side together with outgoing requests and incoming responses. This precisely defined observation is processed by a collector and saved as a file with spans and traces.

Although we cannot directly observe the API, we try to localize an API specification file. The location may be guessed, or derived form response meta-data. Additionally, the API specification file could be provided manually.

The deprecation detector analyzes the collector's results and combines the data with the API specification (and possible other knowledge bases) to decide if the call is to a deprecated API.

a response. If we use the auto-instrumentation of OpenTelemetry, a short instruction and a collector configuration suffice if they are chosen precisely as shown in Listing 11.1.

OpenTelemetry Instrumentation Configuration (System Property)

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

OpenTelemetry Collector Configuration (YAML)

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

Listing 11.1.: OpenTelemetry Configurations for the Hybrid Approach.

The auto-instrumentation is instructed to capture response headers for *Deprecation*, *Sunset*, and *Link* tags. URL information in requests is recorded automatically and does not require an explicit instruction. The collector is instructed to export all raw data into a JSON file.

11.2.3 Detection

Detection from Response Meta-Data First, we examine the response's meta-data. If deprecation or sunset header fields are set, we have a strong indication for deprecation of the respective API element. Second, we try to discover a suitable link from the response meta-data that points to an OpenAPI description.

Detection from OpenAPI Documents Using the link to an OpenAPI description provided by the previous detection attempt or an OpenAPI description received with other means (e. g., searching standard locations or using a manually provided document), the deprecation tag and the textual description for the called element are checked.

This step requires additional attention to the URL to verify that the OpenAPI description actually describes the correct API. While the URL in the HTTP request always denotes the precise call target correctly, splitting this URL into parts often allows multiple options. For the example URL

https://time.example/api/v2/timeservice?placeid=germany/hamburg

the path given in the OpenAPI description may either be

/timeservice

(using `time.example/api/v2` as the server URL) or

/v2/timeservice

(using `time.example/api/` as the server URL). To mitigate this ambiguity, we search for suitable elements for all splitting possibilities.

Detection from Other Sources We further include a general connection to an arbitrary knowledge base. In this abstraction, every service that may provide the deprecation status of a submitted URL by returning `true` or `false` is considered a knowledge base for this scenario. This may actually be a database holding such information or another tool determining the deprecation status on demand, e. g., using machine learning, which we did not cover explicitly in our discussions.

11.2.4 Report

In a first step, the results from the detection are written into a JSON file. From here multiple options arise, that include but are not limited to the following.

IDE plugin Reproducing the behavior of local API deprecation checks, one option is to provide the detection results in the client developer's IDE. A prototype to present results from a deprecation analysis was developed in (Nitz, 2022). This implementation uses the PMD analysis tool as a base.

However, the IDE plugin is in general not a suitable option for this variation of the detection framework: The detection is performed dynamically and possibly repeatedly with each execution, but the reporting would be delivered only to the developer if they are inspecting or writing code with their IDE. This limits the usability of the detection framework essentially. In particular, for cases in which deprecation is introduced later.

Deprecation Analysis as Part of Continuous Integration If a continuous integration pipeline is used, the deprecation detection reports from the test executions can be directly returned to the process and be integrated in the result report of the pipeline.

Issue Trackers Following the point of view, that a deprecation warning needs to be checked by a client developer to decide on a suitable reaction, a detected usage of an deprecated API may be issued as a ticket into an issue management software. (Müller, 2024) implemented a proof-of-concept for Atlassian Jira.

This proof of concept showed a necessary additional step: consolidating reports. If no such additional processing step is implemented, the deprecation detector may open a new issue for each call to a deprecated element, even if the same call has already been reported from another execution. Section 14.2 will discuss further options on the reduction of information overflow from the deprecation detection framework.

Other Communication with Developers Depending on the working environments and practices of developers, the detection framework may support further communication channels,e. g., sending an email or a message through a chat service: (Müller, 2024) presents a prototype for a message in the Slack communication tool. Furthermore, the intermediate file allows adaption for the inclusion in other analysis tools.

11.3 Implementation

This section describes the implementation that is provided in (Bonorden, 2026). An earlier version of the approach and a corresponding prototypical implementation have been published in (Bonorden and van Hoorn, 2024a) with the code for the detection framework available at (Bonorden and van Hoorn, 2024b).

This prototype features the architectural parts as distinct components that may be executed independently of each other: a collector and a detector.

For the collector, the configuration from Listing 11.1 is used without any changes. Thus, the auto-instrumentation is used for all instrumentable clients, and the results from the observation are exported to a local file `output.json`.

The second component, the detector, subsequently reads the local file containing the exported observation. It then iterates over all recorded spans. If a deprecation or sunset header field is detected, the result of this check is saved. If a link pointing possibly to an OpenAPI description is detected, the detector tries to download such file. In addition, a local directory is checked for a suitable OpenAPI document. If an OpenAPI description is found, the detector searches for deprecation information in the elements deprecated tag and textual description. Finally, also a knowledge base is checked. For the prototype this is another locally executable Java program

offering one HTTP endpoint that accepts a URL in the body of a post request and returns either true or false.

Following the detection, the report is written to the local file system as a JSON file. As discussed before, from here multiple options are possible. However the report does not affect the core functionality of the deprecation detection framework. An example from an exported file is shown in Listing 11.2.

JSON

```
1  {
2      "URL": "https://time.example/timeservice?placeId=germany/hamburg",
3      "httpMethod": "GET",
4      "apiSpecification": "time-example.json",
5      "deprecated": true,
6      "deprecationInformation": [
7          "Sunset field in response header: Thu, 31 Dec 2026 23:59:59 GMT",
8          "Operation deprecated in API specification:
           ↪  specifications/time-example.json"
9      ]
10 }
```

Listing 11.2.: A Result from the Prototype of the Deprecation Detection Framework.

This result shows that the detector has analyzed a call to the URL given in line 2 and found an API specification that was saved locally under the name shown in line 3. The analysis shows that both, response meta-data (line 7) and OpenAPI description (line 8) feature deprecation information.

11.4 Discussion

We have shown that a hybrid approach combining the advantages of both, static and dynamic analyses, is possible, by engineering a concept and implementing a prototype. This provides an answer to research question RQ 3 before we will evaluate concept and implementation empirically in the subsequent chapter.

Answer to RQ 3 *How can a framework be designed and implemented to improve the detection of deprecated remote API usage through language-independent deprecation analysis of dynamic calls and multi-source data integration?*

A hybrid detection framework can utilize aspects of both, static and dynamic analysis. The core elements of the detection framework are executed

dynamically. In particular, the detection on the client side for HTTP calls and their targets is performed solely dynamically. In contrast, the detection on the provider side utilizes dynamic analyses (i. e., of response meta-data) and static analyses (i. e., of OpenAPI descriptions). Despite the dynamic execution, the analysis may be statically prepared, e. g., by providing OpenAPI descriptions for the endpoints that are known to be called.

Evaluation

12

To evaluate the described concept and its prototypical evaluation, we discuss the adherence to the stated design principles and design and conduct an empirical evaluation.

12.1 Adherence to Design Principles

In Section 12.1, we have derived four design principles from the preceding parts of this dissertation. We now find that our deprecation detection framework satisfies all of these:

Adherence to DP 1 *The client analysis should not be limited to a single programming language or ecosystem, or to a single communication client library.*

While the instrumentation is specific to the HTTP clients and the environment, in particular, to the programming language, the approach itself is agnostic to language and environment. For a variety of programming languages and HTTP clients, auto-instrumentation is available. For other situations, manual instrumentation is always possible.

Adherence to DP 2 *The client analysis should include call targets that are only available or determined at runtime.*

The deprecation detection framework utilizes dynamic analysis to record calls and their targets. This easily allows the inclusion of targets that may be not known statically. The synthetic evaluation project for the prototypical implementation contains an example that determines the call target from a previous call.

Adherence to DP 3 *The deprecation analysis should include the latest available deprecation information from all relevant sources.*

With the combination of response meta-data, OpenAPI descriptions, and additional knowledge bases, the deprecation detection framework combines the most relevant sources of deprecation information. Furthermore, it records the metadata for the specific call under investigation, downloads available OpenAPI descriptions at runtime, and queries the additional knowledge bases not earlier than the actual analysis. Thus, in summary, the approach utilizes all relevant sources in the latest versions.

Adherence to DP 4 *The deprecation analysis should not require additional prior knowledge about the specific API.*

At no point of the detection activity, prior provided knowledge about the API, e. g., about possible call targets, is needed. It is only an additional option to provide an OpenAPI document that should be included in the analysis.

12.2 Research Design of an Empirical Evaluation

We now aim to study research question RQ 4:

> **RQ 4** **How precise and sensitive is the analysis framework in detecting deprecated remote API usage?**

To enable this investigation, we design an empirical evaluation. However, such an quantitative empirical evaluation of a prototype in this state can only provide insights on the technical core of the problem. It is impossible to reason about the actual usefulness in real-world projects from this data alone. Furthermore, we note that we cannot compare the results against another approach as this is the first.

We design the evaluation setup as a data experiment (Stol and Fitzgerald, 2018). First, this implies using a contrived setting instead of a natural one. We choose this approach, since natural settings require the application in a real-world scenario which is not suitable for the first technical evaluation of the initial prototype to a new problem even if the underlying socio-technical setting would involve human participation (Storey et al., 2020; Ralph et al., 2020). Second, a data experiment should be reproducible. Thus, we do not use real-world API endpoints for the evaluation as their deprecation status may change between executions. Instead we use a locally executable reference project designed for empirical evaluation and specifically constructed evaluation systems containing all of the relevant cases identified in this dissertation.

A similar evaluation of an earlier version of the prototype has been part of (Bonorden and van Hoorn, 2024a) as published in data via (Bonorden and van Hoorn, 2024b). The main difference in the projects that the evaluation is based on is the extension to more HTTP clients in the client-server project. This follows the results from the study (Bonorden, 2025a) presented in Chapter 5.

We use three distinct software systems for the evaluation:

client-server This project is constructed for the purpose of this evaluation. While it is not modeled after real-world systems, it is curated to cover many relevant HTTP clients that perform a call to an API that is deprecated with an unambiguous documentation.

Technically, the project comprises a server (using the Spring framework) and a client (without any framework). The client calls each available API element on the server, and the deprecation framework is expected to issue a warning for every call. The warnings can be analyzed to calculate precision and recall.

Petstore The Petstore project is provided as an example for OpenAPI specifications providing both, a client and a server. However, the server does not deprecate any API elements. Thus, we manually change the deprecation status for multiple operations on the server. We then execute server and client and observe deprecation warnings. Subsequently, we can determine precision and recall on this project.

Lakeside Mutual The Lakeside Mutual project is a fictional project for an insurance company (Zimmermann et al., 2023). The project features a design following the Domain-Driven Design methodology and an implementation using a microservice architecture. Since the project does not feature any deprecated API elements, we manually deprecate a subset. To record suitable test data, we started the services and performed manual actions via the provided web interface.

12.3 Results for the Empirical Evaluation

client-server This project contains 14 different HTTP clients, from which 13 were correctly detected when calling an deprecated API element. The missed client is the Volley HTTP client for which the auto-instrumentation support by OpenTelemetry has ended. No false positives occurred, yielding a precision of 1.00 and a recall of 0.93.

Petstore For the Petstore project, all calls to deprecated API elements were correctly identified. This yields a precision of 1.00 and a recall of 1.00.

Lakeside Mutual Similar to the Petstore project, the evaluating execution of the framework identifies exactly the deprecated calls present in the code. Thus, also for this project, the deprecation detection framework achieves precision of 1.00 and recall of 1.00.

Results for the Evaluation of Previous Versions Although we performed a new evaluation, we need to recall a false negative from a previous evaluation (Bonorden and van Hoorn, 2024a). In that execution, one call in the Petstore to a deprecated API was not detected. Upon investigation we found that the Apache Client was used in multiple ways, e. g., the traditional call pattern and the fluent style. However, at that time, no auto-instrumentation for the fluent style was available.

This shows that relying on the auto-instrumentation provided by OpenTelemetry remains a risk for the recall. It is neither guaranteed that this auto-instrumentation already provides support for a new client or client version, nor that it still supports an old client or version.

12.4 Threats to Validity for the Empirical Evaluation

As an initial technical evaluation for a tool dedicated to a larger sociotechnical setting, the evaluation is also limited by design: Most importantly, the evaluation does not include any human stakeholder.

Furthermore, the design exposes the following two central biases that can threaten the evaluation's validity.

Representation Bias The evaluation is not only limited to very few projects, but also no assessment can be made whether the projects are representative for the entirety of all Java programs with HTTP clients. It needs to be assumed that the external validity of the evaluation is low. Again, this is clear from the construction of the evaluation – and a conscious choice.

Experimenter Bias The second major threat to validity is experimenter bias. Since we compare the results for each project with the manual analysis of these projects, the correctness of the values for precision and recall depends on the researcher performing the manual detection task. This is a threat to construct validity. We mitigated this threat by additionally performing static analysis to identify spots of probable calls to deprecated APIs before going into manual analysis there.

12.5 Conclusions

We can now answer research question RQ 4:

Answer to RQ 4 *How precise and sensitive is the analysis framework in detecting deprecated remote API usage?*

In the technical evaluation on contrived examples, the deprecation detection framework yields a precision of 1.00 for each project and recall values between 0.93 and 1.00. This shows that the approach is technically suitable for the problem, but no implications on the actual usefulness of this tool in practice can be derived.

Discussion

13

The previous Chapter 12 already presented an evaluation of the deprecation detection framework and its implementation. This chapter now adds a discussion of limitations and generalizability, before picking up the initial research questions and problem statement. Finally, we derive recommendations for client developers, API developers, and researchers.

13.1 Limitations and Generalizability

13.1.1 Limitations of the Implementation

As a prototype, the version used for the evaluation still possesses limitations that are not dictated by the general approach, but this concrete implementation.

Auto-instrumentation The correctness of the deprecation detection depends on the correct instrumentation in client code. In the evaluation we have observed some instances that were not covered correctly by the auto-instrumentation. However, we have no general perspective on the quality of the auto-instrumentation and, thus, cannot reason about its influence on the overall quality of the implementation. In addition to a quantitative evaluation of more cases, a qualitative analysis of unsuccessful cases might help to recommend using or not using the auto-instrumentation in practice.

Informal Deprecation Documentation While the approach allows the inclusion of all kinds of sources for deprecation information, the implementation only realizes this additional source of information via a generic interface to any API. Neither is an adapter implemented that would actually connect another source of information (e. g., adding information extraction from informal sources) nor is a weighted approach implemented to combine the results of multiple sources with differences in credibility.

Alignment Actual Usage of Deprecation Channels The evaluation has been designed with contrived project settings. In addition to a general evaluation in a greater context, a larger study on the usage of deprecation information in response meta-data is necessary to support capturing the correct header fields.

13.1.2 Limitations of the Approach

Although the deprecation detection framework is specifically engineering following broad analysis of relevant aspects, some technical limitations remain in the current version. Section 14.2 will discuss the greater socio-technical context.

Observability The deprecation detection framework relies on the observation of runtime data in the client. In the case of a client in the Java ecosystem with access to all source and bytecode files, the auto-instrumentation by OpenTelemetry is applicable. It remains an open question if this assumption also holds for other execution environments. The access could be limited in a technical way, e. g., a communicating component is available only in a binary file, or the communication is not directly pointed towards the API, but uses an intermediate hub. Furthermore, the access could be limited by security or privacy concerns, e. g., users might not want share execution traces recorded on their smart phones as it may include precise call targets and information about the content of the HTTP calls.

Overhead In the evaluation of dynamic approaches, we relied on results by others regarding the overhead of individual instrumentations and the observation of entire systems. The precise overhead for the deprecation detection framework needs to be studied for projects in practice to derive better assessments of the approach's efficiency.

Coverage For the first step, identifying calls and their targets, we opted for the dynamic analysis. In particular, this improves results for the identification of call targets immensely. However, it remains unclear if the general disadvantages for dynamic analysis are relevant to the effectiveness of the deprecation detection framework. In particular, the amount of missed calls to deprecated APIs, because they have not been executed for a long period of time, is of interest.

13.1.3 Generalizability

The limitations discussed before, cover the technical limitations of the precise setting. Another kind of question for limitations arises if we consider changes to the situation. Therefore, we now go backwards on the setting restrictions introduced in Chapter 2 and discuss the framework's generalizability.

Programming Languages The deprecation detection framework is not limited to the Java programming language. We chose this environment since it features a variety of HTTP clients. Since other programming languages have less ways to use HTTP endpoints, also the deprecation detection is more focused. OpenTelemetry provides auto-instrumentation for the programming languages and ecosystems Go, .NET, PHP, Python, and Javascript. Furthermore it features an instrumentation set for binary executables. For other languages, manual instrumentation may be used. Thus, while the extend of automated support differs, the approach itself is generalizable to other programming languages.

Synchronous Communication Since all styles of synchronous communication are similar, the extensibility of the framework depends on the observability of the communication and the availability of deprecation documentation.

For *GraphQL*, both aspects are fulfilled by the standard format specification and tools. For *SOAP*, OpenTelemetry offers similar observability capacity as for HTTP/REST communication, and the description language WSDL offers an `Obsolete` tag that resembles deprecation.

For *gRPC*, client instrumentation is covered by OpenTelemetry. In the setting of gRPC, the provision of interface description is mandatory, and this description may contain deprecated tags. Therefore the deprecation detection framework is applicable. However, due to the automated comparison of messages and services against their specification, in many cases, the deprecation is already reported without the need for an additional framework.

For *CORBA*, little support is offered by the OpenTelemetry tools. Also, the CORBA interface description language does not provide an option to signal deprecation. However, other observability frameworks (e. g., IBM Instana) feature CORBA better and the interface can be extended by a read-only attribute named `deprecated`. Thus, the abstract deprecation detection framework is also applicable for this communication style.

Event-based Communication In event-based communication, the setup of request and response between a client and a provider is no longer present. Thus, the original question that can be phrased as “Should I call this method?” cannot be applied to the setting. Instead we could ask “Should I still send this event?” or “Is it possible that I receive this event anytime?”. Both questions require an answer that cannot be provided by the deprecation detection framework. The first question needs to be addressed to a schema registry, and the second question can universally be answered with “Yes”, since participants in an even-based architecture

are expected to be backwards-compatible even if a new schema version arises.

However, similar methodical approaches can be used to analyze this setting statically (Singh and Koziolek, 2024) and dynamically (Singh et al., 2022).

Access-based Connectors For communication through shared data access, the option to signal deprecation depends on the technology used to store the shared data. While a database system may offer an option to deprecate a table, reading and writing in a text file would require an individual convention for deprecation. Thus, the deprecation detection framework is not applicable in such settings.

Conclusion We have found the deprecation detection framework to be generalizable to other programming languages and to other styles of synchronous communication. However, each of these settings requires an individual configuration of auto-instrumentation and a specific implementation for the detector component.

13.2 Research Questions and Objective

13.2.1 Answers to the Research Questions

For an overview of the entire dissertation, we compile the answers for the five main research questions:

Research Question RQ 0 *What is the state of research on the deprecation of local and remote APIs?*

Research on API deprecation mainly addresses human stakeholders, provides descriptive analyses as well as constructive solutions, and applies data research strategies. Furthermore, research focuses Java and Android ecosystems, and concentrates on clients' reaction to deprecation. Research gaps remain for the diversity of API types (in particular, remote APIs), for an integral view on the entire process (in particular, joining views of providers and clients, and methodical triangulation) and for an extension to root causes.

Research Question RQ 1 *How can existing static analysis techniques for local APIs be adapted to detect usage of deprecated remote APIs?*

For all essential steps in the generalized deprecation detection process a static analysis technique for remote APIs is available. However, their effectiveness is clearly limited due to the nature of remote APIs using dynamically determined targets and also a possibly dynamically changing deprecation status.

Research Question RQ 2 *Which runtime properties of remote API calls are relevant to identify usage of deprecated remote APIs?*

For all essential steps in the generalized deprecation detection process a dynamic analysis technique for remote APIs is available. The main limitation for this approach arises in the rare usage of deprecation information in response meta-data. Another limitation is presented by depending on auto-instrumentation, but this is rather minor.

Research Question RQ 3 *How can a framework be designed and implemented to improve the detection of deprecated remote API usage through language-independent deprecation analysis of dynamic calls and multi-source data integration?*

A hybrid detection framework can utilize aspects of both, static and dynamic analysis. The core elements of the detection framework are executed dynamically. In particular, the detection on the client side for HTTP calls and their targets is performed solely dynamically. In contrast, the detection on the provider side utilizes dynamic analyses (i. e., of response meta-data) and static analyses (i. e., of OpenAPI descriptions). Despite the dynamic execution, the analysis may be statically prepared, e. g., by providing OpenAPI descriptions for the endpoints that are known to be called.

Research Question RQ 4 *How precise and sensitive is the analysis framework in detecting deprecated remote API usage?*

In the technical evaluation on contrived examples, the deprecation detection framework yields a precision of 1.00 for each project and recall values between 0.93 and 1.00. This shows that the approach is technically suitable for the problem, but no implications on the actual usefulness of this tool in practice can be derived.

13.2.2 Accomplishments towards the Objective

Following the five research questions, we come back to the initial objective, given as the following problem statement:

Improve	the detection of deprecated remote API usage
by	designing and implementing a framework
that	provides language-independent deprecation analysis of dynamic calls and integrates multiple data sources
in order to	enable client developers to effectively react to evolving ecosystems.

We now have achieved this objective in a technical and positivistic way. In comparison to the prior state of no applicable framework for this task, the outcome is obviously an improvement. However, from a pragmatic point of view, we could not verify the last part of the problem statement as we did not observe the frameworks application in a real environment, in particular an evolving technical ecosystem. Section 14.2 will discuss possible next steps to complete the accomplishment also for the pragmatic interpretation of the task.

13.3 Recommendations

Following the overall problem analysis and solution construction, we can now derive recommendations for roles involved with the deprecation of remote APIs.

Recommendations for API Clients API clients should observe deprecation information on all relevant channels. Since these channels take diverse forms, a systematic approach is necessary. This includes reviewing HTTP response header meta-data and consulting OpenAPI descriptions. If the detection framework is applied, this can be conducted automatically. In other cases, a recurring manual analysis is necessary.

Furthermore, client developers should actively manage static dependencies in their project. Dependencies that are never updated or not actively used increase the software's security risk and other quality criteria.

Recommendations for API Providers API providers should provide deprecation information in all formats and on all channels available. In doing so, developers of an API should favor technical communication channels, e. g., response metadata and OpenAPI descriptions, in contras to non-technical channels, e. g., a blog post.

Recommendations for Researchers Researchers investigating remote APIs should be aware of the empirical insights from this study. In particular, few Java HTTP clients are used very often, and all other clients are rarely used. Furthermore, call targets cannot be reliably extracted by static analysis.

Furthermore, they may utilize the general models from Part I to guide their future research or use the deprecation detection framework as a basis for further research or as another solution to compare to.

14 Conclusions

14.1 Summary

This dissertation has been concerned with the engineering of an detection framework for usage of deprecated remote APIs. While the role of deprecation in local ecosystems has been a guiding model from the start, the exact transferability had to be studied.

In preparation, we first defined two models of deprecation: a four-step model of the deprecation process spanning clients and providers, and a three step model for the generalized detection of deprecation (Chapters 2 and 3).

Subsequently, we systematically reviewed research on API deprecation to discover the lack thereof for remote APIs (Chapter 3).

We then moved on to analyze the current state for HTTP clients in Java and deprecation information in OpenAPI descriptions, and discussed their use for static (Part II) and dynamic (Part III) analysis.

Following the investigations, we combined the insights to engineer the deprecation detection framework and implement a prototype (Chapter 11). Subsequently, we evaluated the approach and discussed it (Chapters 12 and 13).

The result is a thoughtfully designed deprecation detection framework that proves the technical feasibility of such an endeavor. The result may act as basis for a practical software solution which needs to be developed and evaluated in the context of software projects in practice.

14.2 Future Research Directions

While the deprecation detection framework has been defined and a prototype has been implemented, future work may add upon these achievements. This final section shortly surveys some possible directions for future research.

Technical Adjustments for this Prototype The purpose of the prototype has been to show the technical feasibility of the deprecation detection framework. As such, the implementation opted in favor of a lightweight approach accepting tradeoffs for robustness, maintainability and optimization. Future advancements on this level may address the re-engineering of essential structures and implementation details, e. g., to implement an advanced internal management of known OpenAPI documents.

Evaluation in Context On the level of the general deprecation detection framework, the next step should be the evaluation in a greater socio-technical context in practice, to further include actual needs and human perspectives. In such a setting, the prototype needs to be seen as a work object and not as a first suggestion for a final product. In particular, future work needs to work on possibilities for the third step of the generalized deprecation detection process: the presentation of findings. This aspect needs to be tailored to needs and environments in practice and evaluated there.

Management of Warnings Once the reporting of findings has been added to a setting in practice, it needs to be discussed what should be presented through the identified channels. While deprecation detection for local APIs works evaluates each defined call once, the deprecation detection framework for remote APIs evaluates each execution of each possible call. This may lead to a large number of the same or similar deprecation warning. Thus, a management component of warnings needs to be introduced to prevent alarm fatigue Ostberg and Wagner, 2016. Such a management might include consolidation of reoccurring deprecation information into a single warning, or sampling the executions to actually include in the analysis.

Support for Automated Refactoring To foster a better maintenance of OpenAPI descriptions and response meta-data, additional features may help. One option is to offer instructions for an automated adaption to a changed API. A client could then automatically apply provided transformation rules throughout the same codebase.

Integration of AI Methods As a final extension direction for the framework, AI support needs to be discussed for each step. A user of the framework might choose whether they want to activate or deactivate AI tasks in their specific setup. AI tasks might include the classification of informally provided information, automated web search for OpenAPI descriptions, or automated refactoring based on information for alternatives to a deprecated API element.

Prevention of API Deprecation By studying large amounts of deprecation situations, the provider's reasons and the client's reactions, the need for deprecation can be understood better and API design can be improved to minimize the need for deprecation, e. g., using architectural options or anticipating possible changes.

Extending the Understanding through Additional Empirical Studies

Finally, all aspects that were concerned with understanding the usage of deprecated remote APIs may be empirically studied further to guide further practical advancements. Examples include interviews for the use of specific HTTP clients in Java, reasons to (not) use deprecation-related fields in OpenAPI or in HTTP response data.

Appendices

Publications from the Dissertation Project

This chapter lists six academic contributions to conferences and workshops that present parts of the dissertation:

- Leif Bonorden (2019). "Interfaces in Modular Software Systems: Some Research Questions". In: *21. Workshop Software-Reengineering und -Evolution (WSRE 2019)*. Vol. 39/2. Softwaretechnik-Trends, pp. 23–24
- Leif Bonorden and Matthias Riebisch (2022a). "API Deprecation: A Systematic Mapping Study". In: *48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2022)*, pp. 451–458. DOI: 10.1109/seaa56994.2022.00076
- Achim Röll and Leif Bonorden (2022). "Improving API Design Skills with the API Design Fest: Insights from a Preliminary Experiment with Students". In: *24. Workshop Software-Reengineering und -Evolution (WSRE 2022)*. Vol. 42/2. Softwaretechnik-Trends, pp. 51–52
- Leif Bonorden and André van Hoorn (2024a). "Detecting Usage of Deprecated Web APIs via Tracing". In: *21st International Conference on Software Architecture (ICSA 2024)*, pp. 23–33. DOI: 10.1109/icsa59870.2024.00011
- Leif Bonorden (2025a). "An Empirical Analysis on the Use of Third-Party HTTP Clients in Open-Source Java Projects". In: *51st Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2025)*. Vol. 16083. Lecture Notes in Computer Science, pp. 361–371. DOI: 10.1007/978-3-032-04207-1_24
- Marc Zimmermann et al. (2026). "Migrating HTTP Client Libraries in Java". In: *28. Workshop Software-Reengineering und -Evolution (WSRE 2026)*. Vol. 46/2. Softwaretechnik-Trends

Conferences

Three conference publications are related to the contents of this thesis:

Leif Bonorden and Matthias Riebisch

API Deprecation: A Systematic Mapping Study

48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2022), 2022

Abstract Application Programming Interfaces (APIs) are the prevalent interaction method for software modules, components, and systems. As systems and APIs evolve, an API element may be marked as deprecated, indicating that its use is disapproved or that the feature will be removed in an upcoming version. Consequently, deprecation is a means of communication between developers and, ideally, complemented by further documentation, including suggestions for the developers of the API's clients.

API deprecation is a relatively young research area that recently gained traction among researchers. To identify the current state of research as well as to identify open research areas, a meta-study that assesses scientific studies is necessary. Therefore, this paper presents a systematic mapping study on API deprecation to classify the state of the art and identify gaps in the research field. We identified and mapped 36 primary studies into a classification scheme comprising general and API-specific categories. Furthermore, we identified five major gaps in previous research on API deprecation.

Contribution L. Bonorden was responsible for all steps from the formulation of research questions and the decisions on a research design up to the writing of the article. M. Riebisch contributed to the conceptualization, provided a second opinion on subjective choices in the systematic mapping process, and reviewed the manuscript.

Additional Material The artifact (Bonorden and Riebisch, 2022b) contains two files. A list of *excluded* studies presents the criteria for each study that led to exclusion. A list of *included* studies features each primary study with the respective classifications.

Leif Bonorden and André van Hoorn

Detecting Usage of Deprecated Web APIs via Tracing

21st International Conference on Software Architecture (ICSA 2024), 2024

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.

Contribution L. Bonorden was responsible for all steps from the formulation of research questions and the decisions on a research design up to the writing of the article. A. van Hoorn contributed to the initial conceptualization, and to the configuration and application of OpenTelemetry, and reviewed the manuscript.

Additional Material The artifact (Bonorden and van Hoorn, 2024b) contains a prototype implementations of the presented approach as well as evaluation data and the example systems the evaluation was performed on.

The artifact has been submitted as part of the peer-review process for the publication. Furthermore, the artifact has been submitted to the conference's artifact evaluation track. Following peer-review, the artifact has been awarded with *Research Object Reviewed (Functional)*[1] and *Open Research Object*[2] badges.

[1] ROR Functional: "Artifacts documented, consistent, complete, exercisable, and include appropriate evidence of verification and validation."

[2] ORO: "Author-created artifacts relevant to this paper have been placed in a publicly accessible archival repository. A DOI or link to this repository along with a unique identifier for the object is provided."

Leif Bonorden

An Empirical Analysis on the Use of Third-Party HTTP Clients in Open-Source Java Projects

51st Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2025), 2025

Abstract External communication libraries (e. g., for HTTP/REST) are static dependencies that enable dynamic dependencies in software systems. Such third-party HTTP libraries may influence the system's qualities. Thus, the selection out of numerous third-party HTTP clients needs to be taken carefully. However, the use of such clients has not been studied broadly.
We conduct a quantitative empirical study of 18,879 open-source Java repositories and analyze 259,683 configuration files. We further investigate a subset of these repositories qualitatively and in more depth. We have found that Apache HttpClient, Spring Web, and OkHttp are the most used third-party HTTP clients in Java open-source projects. Further analysis has shown that declared dependencies are often not actively managed, reference outdated versions, or are entirely unused.

Contribution L. Bonorden is the sole author of this publication.

Additional Material The artifact (Bonorden, 2025b) contains two software tools to automate parts of the analysis, a `DependencyCollector` and a `DependencyIdentifier`. Furthermore, the artifact comprises empirical data collected automatically and manually during the study.

Workshops

Three workshop contributions related to this dissertation have been published as extended abstracts:

Leif Bonorden

Interfaces in Modular Software Systems: Some Research Questions

21. Workshop Software-Reengineering und -Evolution (WSRE 2019), *2019*

Abstract. Modularity of software systems is well-known and supported by various theories. Interfaces and interactions between such software modules are differently seen and treated from different points of view.

This article briefly surveys semi-formal models, formal specifications and technical implementations, and introduces corresponding research questions regarding the compatibility of these perspectives with each other and their role in the software development process

Achim Röll and Leif Bonorden

Improving API Design Skills with the API Design Fest: Insights from a Preliminary Experiment with Students

24. Workshop Software-Reengineering und -Evolution (WSRE 2022), *2022*

Abstract. APIs should be stable and demand a careful evolution, which requires a good initial design. Such API design skills usually come from experience, but the API Design Fest intends to compress such experience into a dense course. While the original training event was intended for practitioners, we are interested in the applicability in software engineering education. Thus, we conducted a slightly adopted API Design Fest with students and report on initial insights. While we find the overall event and its API design activities suitable, we recommend extended preparation on breaking changes for future API Design Fests with students.

Marc Zimmermann, Leif Bonorden, and Oliver Kopp

Migrating HTTP Client Libraries in Java

28. Workshop Software-Reengineering und -Evolution (WSRE 2026), *2026*

Abstract. In Java, HTTP calls are often performed using third-party libraries. If such an (external) library needs to be replaced by another one, a migration task arises. We migrated multiple Java projects to alternative HTTP clients using manual and automated migration strategies – including LLMs. We present some insights from our migration experiments.

List of Primary Studies and their Classification from the Systematic Mapping Study

The tables on the following pages list the primary studies identified in the systematic meta-research process described in Chapter 3. For each study, the reference is given together with the classification for all criteria. This is a summary of the data published in (Bonorden and Riebisch, 2022b).

Primary Study	Beneficiaries			Contribution				Strategy					API Type						Aspect			
	Human Stakeholders	Technical Systems	Researchers	Analysis	Tool	Method	Classification	Field	Lab	Respondent	Data	Non-empirical	Java	Android	JavaScript	Python	other (static)	REST/OpenAPI	Decision (Supplier)	Documentation (Supplier)	Information (Client)	Reaction (Client)
Arvedahl (2018)		X			X			X									X					X
Brito et al. (2018)			X	X							X		X				X			X		
Cogo et al. (2021))		X		X							X				X					X		X
Haryono et al. (2020)		X			X						X			X								X
Haryono et al. (2021)		X			X						X			X								X
Hora et al. (2018)	X			X							X						X					X
Huang et al. (2018)	X			X	X						X			X								X
Ko et al. (2014)	X			X							X		X							X		
Lamothe and Shang (2018)	X			X				X						X								X
Li et al. (2020)	X			X							X			X								X
Mirian et al. (2019)	X			X		X	X				X						X		X			
Moreno et al. (2017)	X				X					X	X		X							X		

Primary Study	Beneficiaries			Contribution				Strategy					API Type						Aspect			
	Human Stakeholders	Technical Systems	Researchers	Analysis	Tool	Method	Classification	Field	Lab	Respondent	Data	Non-empirical	Java	Android	JavaScript	Python	other (static)	REST/OpenAPI	Decision (Supplier)	Documentation (Supplier)	Information (Client)	Reaction (Client)
Nascimento et al. (2021)	X			X						X	X				X					X	X	X
Nguyen et al. (2010)	X				X						X		X									X
Nishi and Damevski (2020)	X				X						X			X								X
Perkins (2005)	X					X					X		X									X
Qiu et al. (2016)	X			X							X		X									X
Raemaekers et al. (2017)	X			X							X		X							X		
Robbes et al. (2012)	X			X							X						X					X
Samak et al. (2020)	X				X						X		X									X
Sawant et al. (2018a)	X		X	X						X			X						X			X
Sawant et al. (2018c)	X			X							X		X									X
Sawant et al. (2018b)	X		X	X							X		X						X			
Sawant et al. (2019)	X			X						X	X		X									X

Primary Study	Beneficiaries			Contribution				Strategy					API Type						Aspect			
	Human Stakeholders	Technical Systems	Researchers	Analysis	Tool	Method	Classification	Field	Lab	Respondent	Data	Non-empirical	Java	Android	JavaScript	Python	other (static)	REST/OpenAPI	Decision (Supplier)	Documentation (Supplier)	Information (Client)	Reaction (Client)
Scalabrino et al. (2019)	X				X						X			X								X
Spoon (2007)		X				X						X	X							X		
Štrobl and Troníček (2013)	X				X						X		X									X
Thung et al. (2019)	X					X					X			X								X
Thung et al. (2020)	X			X		X					X			X								X
Vadlamani et al. (2021)	X				X						X					X					X	
Wang et al. (2020b)	X		X	X							X					X						X
Wang et al. (2020a)	X		X	X							X					X				X		X
Xi et al. (2019)	X				X						X		X									X
Xu et al. (2020)	X		X	X						X			X			X						X
Yasmin et al. (2020)	X		X	X	X						X							X		X		
Zhou and Walker (2016)	X				X						X		X									X

List of Figures

List of Tables

List of Listings

Bibliography

Manel Abdellatif, Rafik Tighilt, Abdelkarim Belkhir, Naouel Moha, Yann-Gaël Guéhéneuc, and Éric Beaudry (2020). "A multi-dimensional study on the state of the practice of REST APIs usage in Android apps". In: *Automated Software Engineering* 27, pp. 187–228. DOI: 10.1007/s10515-020-00272-9 (cit. on p. 52).

Waleed Ammar et al. (2018). "Construction of the Literature Graph in Semantic Scholar". In: *2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers)*, pp. 84–91. DOI: 10.18653/v1/n18-3011 (cit. on p. 25).

Apostolos Ampatzoglou, Stamatia Bibi, Paris Avgeriou, and Alexander Chatzigeorgiou (2020). "Guidelines for Managing Threats to Validity of Secondary Studies in Software Engineering". In: *Contemporary Empirical Methods in Software Engineering*. Ed. by Michael Felderer and Guilherme Horta Travassos, pp. 415–441. DOI: 10.1007/978-3-030-32489-6_15 (cit. on p. 39).

Apostolos Ampatzoglou, Stamatia Bibi, Paris Avgeriou, Marijn Verbeek, and Alexander Chatzigeorgiou (2019). "Identifying, categorizing and mitigating threats to validity in software engineering secondary studies". In: *Information and Software Technology* 106, pp. 201–230. DOI: 10.1016/j.infsof.2018.10.006 (cit. on p. 39).

Svante Arvedahl (2018). "Introducing Debtgrep: a Tool for Fighting Technical Debt in Base Station Software". In: *1st International Conference on Technical Debt (TechDebt 2018)*, pp. 51–52. DOI: 10.1145/3194164.3194183 (cit. on p. 152).

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer (2016). "QL: Object-oriented Queries on Relational Data". In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Vol. 56. LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2. DOI: 10.4230/LIPICS.ECOOP.2016.2 (cit. on p. 71).

Sebastian Baltes and Paul Ralph (2022). "Sampling in software engineering research: a critical review and guidelines". In: *Empirical Software Engineering* 27, 94. DOI: 10.1007/s10664-021-10072-8 (cit. on p. 54).

Len Bass, Paul Clements, and Rick Kazman (2022). *Software Architecture in Practice*. 4th ed. Addison-Wesley (cit. on p. 12).

Tim Berners-Lee, Larry Masinter, and Mark McCahill (1994). *Uniform Resource Locators (URL)*. RFC 1738. DOI: 10.17487/RFC1738 (cit. on p. 48).

Justus Bogner, Sebastian Kotstein, Daniel Abajirov, Timothy Ernst, and Manuel Merkel (2024a). "RESTRuler: Towards Automatically Identifying Violations of RESTful Design Rules in Web APIs". In: *21st International Conference on Software Architecture (ICSA 2024)*, pp. 123–134. DOI: 10.1109/icsa59870.2024.00020 (cit. on p. 19).

Justus Bogner, Sebastian Kotstein, and Timo Pfaff (2023). "Do RESTful API design rules have an impact on the understandability of Web APIs?" In: *Empirical Software Engineering* 28, 132. DOI: 10.1007/s10664-023-10367-y (cit. on p. 20).

Justus Bogner, Pawel Wójcik, and Olaf Zimmermann (2024b). "How Do Microservice API Patterns Impact Understandability? A Controlled Experiment". In: *21st International Conference on Software Architecture (ICSA 2024)*, pp. 158–169. DOI: 10.1109/icsa59870.2024.00023 (cit. on p. 20).

Leif Bonorden (2019). "Interfaces in Modular Software Systems: Some Research Questions". In: *21. Workshop Software-Reengineering und -Evolution (WSRE 2019)*. Vol. 39/2. Softwaretechnik-Trends, pp. 23–24 (cit. on pp. 145, 149).

– (2025a). "An Empirical Analysis on the Use of Third-Party HTTP Clients in Open-Source Java Projects". In: *51st Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2025)*. Vol. 16083. Lecture Notes in Computer Science, pp. 361–371. DOI: 10.1007/978-3-032-04207-1_24 (cit. on pp. 44, 48, 127, 145, 148).

– (2025b). *Third-Party HTTP Clients in Open-Source Java Projects [Dataset]*. Zenodo. DOI: 10.5281/zenodo.15102293 (cit. on pp. 44, 48, 53–55, 148).

– (2026). *A Framework for Detecting Usage of Deprecated Remote APIs [Research Software]*. Zenodo. DOI: 10.5281/zenodo.20114381 (cit. on pp. 44, 122).

Leif Bonorden and Matthias Riebisch (2022a). "API Deprecation: A Systematic Mapping Study". In: *48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2022)*, pp. 451–458. DOI: 10.1109/seaa56994.2022.00076 (cit. on pp. 18, 22, 24, 29, 40, 44, 145, 146).

– (2022b). *API Deprecation: A Systematic Mapping Study [Dataset]*. Zenodo. DOI: 10.5281/zenodo.5650121 (cit. on pp. 22, 44, 146, 151).

– (2023). "API Deprecation: A Systematic Mapping Study". In: *Software Engineering 2023*. Ed. by Gregor Engels, Regina Hebig, and Matthias Tichy. Vol. P-332. Lecture Notes in Informatics. Gesellschaft für Informatik, pp. 41–42 (cit. on p. 22).

Leif Bonorden and André van Hoorn (2024a). “Detecting Usage of Deprecated Web APIs via Tracing”. In: *21st International Conference on Software Architecture (ICSA 2024)*, pp. 23–33. DOI: 10.1109/icsa59870.2024.00011 (cit. on pp. 18, 43, 94, 102, 122, 127, 129, 145, 147).

– (2024b). *Detecting Usage of Deprecated Web APIs via Tracing: Replication Package*. Zenodo. DOI: 10.5281/zenodo.10250357 (cit. on pp. 44, 122, 127, 147).

Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes (2018). “On the use of replacement messages in API deprecation: An empirical study”. In: *Journal of Systems and Softwware* 137, pp. 306–321. DOI: 10.1016/j.jss.2017.12.007 (cit. on pp. 27, 152).

Filipe Roseiro Cogo, Gustavo Ansaldi Oliva, and Ahmed E. Hassan (2021). “Deprecation of Packages and Releases in Software Ecosystems: A Case Study on NPM”. In: *IEEE Transactions on Software Engineering* 48.7, pp. 2208–2223. DOI: 10.1109/TSE.2021.3055123 (cit. on pp. 36, 152).

Ira W. Cotton and Frank S. Greatorex (1968). “Data structures and techniques for remote computer graphics”. In: *1968 Fall Joint Computer Conference (FJCC)*, pp. 533–544. DOI: 10.1145/1476589.1476661 (cit. on p. 15).

Ozren Dabic, Emad Aghajani, and Gabriele Bavota (2021). “Sampling Projects in GitHub for MSR Studies”. In: *18th International Conference on Mining Software Repositories (MSR 2021)*, pp. 560–564. DOI: 10.1109/msr52588.2021.00074 (cit. on p. 53).

Sanjay Dalal and Erik Wilde (2025). *The Deprecation HTTP Response Header Field*. RFC 9745. DOI: 10.17487/RFC9745 (cit. on p. 103).

Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson (2004). “How a Good Software Practice Thwarts Collaboration – The Multiple Roles of APIs in Software Development”. In: *12th International Symposium on Foundations of Software Engineering (FSE 2004)*, pp. 221–230. DOI: 10.1145/1029894.1029925 (cit. on p. 16).

Fabio Di Lauro, Souhaila Serbout, and Cesare Pautasso (2022). “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. DOI: 10.1007/978-3-031-16697-6_3 (cit. on pp. 40, 80).

Amine El Malki and Uwe Zdun (2023). “Combining API Patterns in Microservice Architectures: Performance and Reliability Analysis”. In: *2023 International Conference on Web Services (ICWS 2023)*, pp. 246–257. DOI: 10.1109/icws60048.2023.00044 (cit. on p. 20).

Amine El Malki, Uwe Zdun, and Cesare Pautasso (2022). “Impact of API Rate Limit on Reliability of Microservices-Based Architectures”. In: *16th International Conference on Service-Oriented System Engineering (SOSE 2022)*, pp. 19–28. DOI: 10.1109/sose55356.2022.00009 (cit. on p. 20).

Michael D. Ernst (2003). “Static and dynamic analysis: snyergy and duality”. In: *1st Workshop on Dynamic Analysis (WODA 2003)*, pp. 25–28 (cit. on pp. 88, 110).

Eric Evans (2003). *Domain-Driven Design. Tackling Complexity in the Heart of Software*. Addison-Wesley (cit. on p. 19).

Roy T. Fielding, Mark Nottingham, and Julian Reschke (2014). *Hypertext Transfer Protocol (HTTP/1.1): Caching*. RFC 7234. DOI: 10.17487/RFC7234 (cit. on p. 103).

– (2022a). *HTTP Caching*. RFC 9111. DOI: 10.17487/RFC9111 (cit. on p. 103).

– (2022b). *HTTP/1.1*. RFC 9112. DOI: 10.17487/rfc9112 (cit. on p. 96).

Pascal Gadient, Mohammad Ghafari, Marc-Andrea Tarnutzer, and Oscar Nierstrasz (2020). “Web APIs in Android through the Lens of Security”. In: *27th International Conference on Software Analysis, Evolution and Reengineering (SANER 2020)*, pp. 13–22. DOI: 10.1109/saner48275.2020.9054850 (cit. on pp. 52, 56, 64).

Yogya Gamage, Nadia Gonzalez Fernandez, Martin Monperrus, and Benoit Baudry (2025). “Software Bills of Materials in Maven Central”. In: *22nd International Conference on Mining Software Repositories (MSR 2025)*, pp. 339–343. DOI: 10.1109/MSR66628.2025.00062 (cit. on p. 52).

Patric Genfer and Uwe Zdun (2021). “Identifying Domain-Based Cyclic Dependencies in Microservice APIs Using Source Code Detectors”. In: *15th European Conference on Software Architecture (ECSA 2021)*, pp. 207–222. DOI: 10.1007/978-3-030-86044-8_15 (cit. on pp. 51, 72).

Sebastian Gerdes, Tobias Fechner, and Matthias Riebisch (2018). “Identification of Technology Features to Understand and Maintain Software Architectures”. In: *25th Australasian Software Engineering Conference (ASWEC 2018)*, pp. 210–214. DOI: 10.1109/aswec.2018.00035 (cit. on p. 71).

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman (2025). *The Java Language Specification – Java SE 25 Edition* (cit. on p. 18).

Yasmeen Hammad, Amro Al-Said Ahmad, and Peter Andras (2025). “An empirical study on the performance overhead of code instrumentation in containerised microservices”. In: *Journal of Systems and Software* 230, 112573. DOI: 10.1016/j.jss.2025.112573 (cit. on p. 98).

Nicolas Harrand, Thomas Durieux, David Broman, and Benoit Baudry (2021). "Automatic Diversity in the Software Supply Chain". In: arXiv: 2111.03154 [cs.SE] (cit. on p. 52).

Stefanus A. Haryono, Ferdian Thung, Hong Jin Kang, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang (2020). "Automatic Android Deprecated-API Usage Update by Learning from Single Updated Example". In: *28th International Conference on Program Comprehension (ICPC 2020)*, pp. 401–405. DOI: 10.1145/3387904.3389285 (cit. on p. 152).

Stefanus A. Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, Hong Jin Kang, Lucas Serrano, and Gilles Muller (2021). "AndroEvolve: Automated Update for Android Deprecated-API Usages". In: *43rd International Conference on Software Engineering (ICSE 2021) – Companion Proceedings*, pp. 1–4. DOI: 10.1109/ICSE-Companion52605.2021.00021 (cit. on pp. 27, 152).

Michael Hausenblas (2024). *Cloud Observability in Action*. Manning Publications (cit. on p. 94).

Kim Hoffmann (2026). "How and why are APIs deprecated?" Master's thesis. Universität Hamburg (cit. on p. 78).

Gregor Hohpe and Bobby Woolf (2013). *Enterprise Integration Patterns. Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley. 683 pp. (cit. on p. 14).

Andre Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stéphane Ducasse (2018). "How do developers react to API evolution? A large-scale empirical study". In: *Software Quality Journal* 26, pp. 161–191. DOI: 10.1007/s11219-016-9344-4 (cit. on p. 152).

Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung (2018). "Understanding and Detecting Callback Compatibility Issues for Android Applications". In: *33rd International Conference on Automated Software Engineering (ASE 2018)*, pp. 532–542. DOI: 10.1145/3238147.3238181 (cit. on p. 152).

Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu (2022). "Characterizing usages, updates and risks of third-party libraries in Java projects". In: *Empirical Software Engineering* 27, 90. DOI: 10.1007/s10664-022-10131-8 (cit. on p. 52).

Richard Hutcheson, Austin Blanchard, Noah Lambaria, Jack Hale, David Kozak, Amr S. Abdelfattah, and Tomas Cerny (2024). "Software Architecture Reconstruction for Microservice Systems Using Static Analysis via GraalVM Native Image". In: *31st International Conference on Software Analysis, Evolution and Reengineering (SANER 2024)*. DOI: 10.1109/SANER60148.2024.00008 (cit. on p. 51).

ISO/IEC 2382 (2015). *Information technology — Vocabulary*. Standard. International Organization for Standardization (cit. on p. 14).

ISO/IEC 25010 (2023). *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model*. Standard. International Organization for Standardization (cit. on pp. 13, 14).

Stefanie Jasser (2020). "Enforcing Architectural Security Decisions". In: *17th International Conference on Software Architecture (ICSA 2020)*, pp. 35–45. DOI: 10.1109/icsa47634.2020.00012 (cit. on p. 51).

JEP 110 (2014). *HTTP/2 Client (Incubator)*. JDK Enhancement Proposal (cit. on p. 50).

JEP 321 (2017). *HTTP Client API*. JDK Enhancement Proposal (cit. on pp. 48, 50).

JEP 517 (2022). *HTTP/3 for the HTTP Client API*. JDK Enhancement Proposal (cit. on p. 50).

Mehdi Keshani, Gideon Bot, Priyam Rungta, Maliheh Izadi, Arie Van Deursen, and Sebastian Proksch (2024). "Maven Unzipped: Exploring the Impact of Library Packaging on the Ecosystem". In: *40th International Conference on Software Maintenance and Evolution (ICSME 2024)*, pp. 50–62. DOI: 10.1109/icsme58944.2024.00016 (cit. on p. 52).

Barbara Kitchenham (2004). *Procedures for Performing Systematic Reviews*. Keele University Technical Report TR/SE-0401. Keele University and National ICT Australia (cit. on p. 23).

Barbara Kitchenham and Stuart Charters (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering*. EBSE Technical Report EBSE-2007-01. Keele University and University of Dunham (cit. on pp. 23, 25).

Barbara A. Kitchenham, David Budgen, and O. Pearl Brereton (2011). "Using mapping studies as the basis for further research – A participant-observer case study". In: *Information and Software Technology* 53.6, pp. 638–651. DOI: 10.1016/j.infsof.2010.12.011 (cit. on p. 24).

Barbara A. Kitchenham, Tore Dybå, and Magne Jørgensen (2004). "Evidence-based Software Engineering". In: *26th International Conference on Software Engineering (ICSE 2024)*, pp. 273–281. DOI: 10.1109/icse.2004.1317449 (cit. on p. 23).

Deokyoon Ko, Kyeongwook Ma, Sooyong Park, Suntae Kim, Dongsun Kim, and Yves Le Traon (2014). "API Document Quality for Resolving Deprecated APIs". In: *21st Asia-Pacific Software Engineering Conference (APSEC 2014)*, pp. 27–30. DOI: 10.1109/APSEC.2014.87 (cit. on p. 152).

Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang (2022). "A Systematic Review of API Evolution Literature". In: *ACM Computing Surveys* 54.8, 171. DOI: 10.1145/3470133 (cit. on pp. 15, 20).

Maxime Lamothe and Weiyi Shang (2018). “Exploring the Use of Automated API Migrating Techniques in Practice: An Experience Report on Android”. In: *15th International Conference on Mining Software Repositories (MSR 2018)*, pp. 503–514. DOI: 10.1145/3196398.3196420 (cit. on p. 152).

Alexander Lercher, Johann Glock, Christian Macho, and Martin Pinzger (2024). “Microservice API Evolution in Practice: A Study on Strategies and Challenges”. In: *Journal of Systems and Software* 215, 112110. DOI: 10.1016/j.jss.2024.112110 (cit. on pp. 15, 40).

Ding Li, Yingjun Lyu, Mian Wan, and William G. J. Halfond (2015). “String Analysis for Java and Android Applications”. In: *10th Joint Meeting on Foundations of Software Engineering (FSE 2015)*, pp. 661–672. DOI: 10.1145/2786805.2786879 (cit. on p. 74).

Li Li, Jun Gao, Tegawendé F. Bissyandé, Lei Ma, Xin Xia, and Jacques Klein (2020). “CDA: Characterising Deprecated Android APIs”. In: *Empirical Software Engineering* 25, pp. 2058–2098. DOI: 10.1007/s10664-019-09764-z (cit. on pp. 27, 152).

Caroline Lima and Andre Hora (2020). “What are the characteristics of popular APIs? A large-scale study on Java, Android, and 165 libraries”. In: *Software Quality Journal* 28.2, pp. 425–458. DOI: 10.1007/s11219-019-09476-z (cit. on p. 16).

Lei Liu, Mehdi Bahrami, Junhee Park, and Wei-Peng Chen (2020). “Web API Search: Discover Web API and Its Endpoint with Natural Language Queries”. In: *18th International Conference on Web Services (ICWS 2020)*, pp. 96–113. DOI: 10.1007/978-3-030-59618-7_7 (cit. on p. 15).

Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker (2019). “Interface Evolution Patterns — Balancing Compatibility and Extensibility across Service Life Cycles”. In: *24th European Conference on Pattern Languages of Programs (EuroPLop 2019)*, 15. DOI: 10.1145/3361149.3361164 (cit. on p. 20).

Shang-Pin Ma, I-Hsiu Liu, Chun-Yu Chen, Jiun-Ting Lin, and Nien-Lin Hsueh (2019). “Version-Based Microservice Analysis, Monitoring, and Visualization”. In: *26th Asia-Pacific Software Engineering Conference (APSEC 2019)*, pp. 165–172. DOI: 10.1109/apsec48747.2019.00031 (cit. on p. 64).

Walid Maalej and Martin P. Robillard (2013). “Patterns of Knowledge in API Reference Documentation”. In: *IEEE Transactions on Software Engineering* 39.9, pp. 1264–1282. DOI: 10.1109/tse.2013.12 (cit. on p. 19).

Neil Madden (2021). *API Security in Action*. Manning (cit. on p. 20).

Charity Majors, Liz Fong-Jones, and George Miranda (2022). *Observability Engineering. Achieving Production Excellence*. O’Reilly (cit. on p. 94).

Ruchika Malhotra and Megha Khanna (2016). “An exploratory study for software change prediction in object-oriented systems using hybridized techniques”. In: *Automated Software Engineering* 24.3, pp. 673–717. DOI: 10.1007/s10515-016-0203-0 (cit. on p. 37).

Michael Meng, Stephanie Steinhardt, and Andreas Schubert (2018). “Application Programming Interface Documentation: What Do Software Developers Want?” In: *Journal of Technical Writing and Communication* 48.3, pp. 295–330. DOI: 10.1177/0047281617721853 (cit. on pp. 15, 19).

Isomi M. Miake-Lye, Susanne Hempel, Roberta Shanman, and Paul G. Shekelle (2016). “What is an evidence map? A systematic review of published evidence maps and their definitions, methods, and products”. In: *Systematic Reviews* 5, 28. DOI: 10.1186/s13643-016-0204-x (cit. on p. 23).

Ariana Mirian, Nikunj Bhagat, Caitlin Sadowski, Adrienne Porter Felt, Stefan Savage, and Geoffrey M. Voelker (2019). “Web Feature Deprecation: A Case Study for Chrome”. In: *41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2019)*, pp. 302–311. DOI: 10.1109/ICSE-SEIP.2019.00044 (cit. on pp. 29, 152).

Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora (2017). “ARENA: An Approach for the Automated Generation of Release Notes”. In: *IEEE Transactions on Software Engineering* 43.2, pp. 106–127. DOI: 10.1109/TSE.2016.2591536 (cit. on p. 152).

Marvin Müller (2024). “Erkennung von Aufrufen an deprecated Web-APIs in HTTP-Clients”. Available at: https://edoc.sub.uni-hamburg.de/informatik/volltexte/2025/311/. Bachelor’s thesis. Universität Hamburg (cit. on pp. 94, 99, 105, 122).

Lauren Murphy, Mary Beth Kery, Oluwatosin Alliyu, Andrew Macvean, and Brad A. Myers (2018). “API Designers in the Field: Design Practices and Challenges for Creating Usable APIs”. In: *2018 Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 249–258. DOI: 10.1109/vlhcc.2018.8506523 (cit. on pp. 19, 37).

Romulo Silva do Nascimento, Eduardo Figueiredo, and Andre Hora (2021). “JavaScript API Deprecation Landscape: A Survey and Mining Study”. In: *IEEE Software* 39.3, pp. 96–105. DOI: 10.1109/MS.2021.3103134 (cit. on pp. 36, 153).

Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen (2010). “A Graph-based Approach to API Usage Adaptation”. In: *25th Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*, pp. 302–321. DOI: 10.1145/1869459.1869486 (cit. on p. 153).

Manziba Akanda Nishi and Kostadin Damevski (2020). “Automatically identifying valid API versions for software development tutorials on the Web”. In: *Journal of Software: Evolution and Process* 32.4, e2227. DOI: 10.1002/smr.2227 (cit. on p. 153).

Thore Nitz (2022). “Erkennung von HTTP-Requests in Java durch statische Analyse von Abstract Syntax Trees”. Bachelor’s thesis. Universität Hamburg (cit. on pp. 73, 121).

Mark Nottingham (2019). *Well-Known Uniform Resource Identifiers (URIs)*. RFC 8615. DOI: 10.17487/RFC8615 (cit. on p. 82).

Jan-Peter Ostberg and Stefan Wagner (2016). “At Ease with Your Warnings: The Principles of the Salutogenesis Model Applied to Automatic Static Analysis”. In: *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, pp. 629–633. DOI: 10.1109/saner.2016.63 (cit. on p. 141).

Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee (2007). “A Design Science Research Methodology for Information Systems Research”. In: *Journal of Management Information Systems* 24.3, pp. 45–77. DOI: 10.2753/MIS0742-1222240302 (cit. on p. 43).

Jeff H. Perkins (2005). “Automatically Generating Refactorings to Support API Evolution”. In: *6th Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*. ACM, pp. 111–114. DOI: 10.1145/1108792.1108818 (cit. on pp. 29, 153).

Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson (2008). “Systematic Mapping Studies in Software Engineering”. In: *12th International Conference on Evaluation and Assessment in Software Engineering (EASE 2008)*, pp. 68–77 (cit. on pp. 22, 23, 25).

Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz (2015). “Guidelines for conducting systematic mapping studies in software engineering: An update”. In: *Information and Software Technology* 64, pp. 1–18. DOI: 10.1016/j.infsof.2015.03.007 (cit. on pp. 23, 33).

Mai T. Pham, Andrijana Rajić, Judy D. Greig, Jan M. Sargeant, Andrew Papadopoulos, and Scott A. McEwen (2014). “A scoping review of scoping reviews: advancing the approach and enhancing the consistency”. In: *Research Synthesis Methods* 5.4, pp. 371–385. DOI: 10.1002/jrsm.1123 (cit. on p. 23).

Ilaria Pigazzini, Francesca Arcelli Fontana, Valentina Lenarduzzi, and Davide Taibi (2020). “Towards Microservice Smells Detection”. In: *3rd International Conference on Technical Debt (TechDebt 2020)*, pp. 92–97. DOI: 10.1145/3387906.3388625 (cit. on p. 64).

Joshua S. Ponelat and Lukas L. Rosenstock (2022). *Designing APIs with Swagger and OpenAPI*. Manning (cit. on p. 78).

Dong Qiu, Bixin Li, and Hareton Leung (2016). "Understanding the API usage in Java". In: *Information and Software Technology* 73, pp. 81–100. DOI: 10.1016/j.infsof.2016.01.011 (cit. on p. 153).

Mikko Raatikainen, Elina Kettunen, Ari Salonen, Marko Komssi, Tommi Mikkonen, and Timo Lehtonen (2021). "State of the Practice in Application Programming Interfaces (APIs): A Case Study". In: *15th European Conference on Software Architecture (ECSA 2021)*, pp. 191–206. DOI: 10.1007/978-3-030-86044-8_14 (cit. on pp. 15, 16, 35).

Florian Rademacher, Sabine Sachweh, and Albert Zündorf (2020). "A Modeling Method for Systematic Architecture Reconstruction of Microservice-Based Software Systems". In: *25th International Conference on Exploring Modeling Methods for Systems Analysis and Development (EMMSAD 2020)*, pp. 311–326. DOI: 10.1007/978-3-030-49418-6_21 (cit. on p. 64).

Steven Raemaekers, Arie van Deursen, and Joost Visser (2017). "Semantic versioning and impact of breaking changes in the Maven repository". In: *Journal of Systems and Software* 129, pp. 140–158. DOI: 10.1016/j.jss.2016.04.008 (cit. on p. 153).

Paul Ralph and Sebastian Baltes (2022). "Paving the Way for Mature Secondary Research: The Seven Types of Literature Review". In: *30th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*, pp. 1632–1636. DOI: 10.1145/3540250.3560877 (cit. on p. 23).

Paul Ralph et al. (2020). "Empirical Standards for Software Engineering Research". In: arXiv: 2010.03525 [cs.SE] (cit. on pp. 39, 52, 63, 127).

Marianna Rapoport, Philippe Suter, Erik Wittern, Ondrej Lhotak, and Julian Dolby (2017). "Who You Gonna Call? Analyzing Web Requests in Android Applications". In: *14th International Conference on Mining Software Repositories (MSR 2017)*, pp. 80–90. DOI: 10.1109/msr.2017.11 (cit. on pp. 52, 64).

Santosh S. Rathore and Sandeep Kumar (2017). "A study on software fault prediction techniques". In: *Artificial Intelligence Review* 51.2, pp. 255–327. DOI: 10.1007/s10462-017-9563-5 (cit. on p. 37).

David Georg Reichelt, Lubomír Bulej, Reiner Jung, and André van Hoorn (2024). "Overhead Comparison of Instrumentation Frameworks". In: *15th ACM/SPEC International Conference on Performance Engineering (ICPE 2024)*, pp. 249–256. DOI: 10.1145/3629527.3652269 (cit. on p. 98).

Mark Richards and Neal Ford (2020). *Fundamentals of Software Architecture. An Engineering Approach*. O'Reilly (cit. on p. 12).

Michael Riegler, Johannes Sametinger, Michael Vierhauser, and Manuel Wimmer (2023). "A model-based mode-switching framework based on security vulnerability scores". In: *Journal of Systems and Software* 200, 111633. DOI: 10.1016/j.jss.2023.111633 (cit. on p. 52).

Romain Robbes, Mircea Lungu, and David Röthlisberger (2012). "How Do Developers React to API Deprecation? The Case of a Smalltalk Ecosystem". In: *20th International Symposium on the Foundations of Software Engineering (FSE 2012)*. DOI: 10.1145/2393596.2393662 (cit. on p. 153).

Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford (2013). "Automated API Property Inference Techniques". In: *IEEE Transactions on Software Engineering* 39.5, pp. 613–637. DOI: 10.1109/tse.2012.63 (cit. on p. 15).

Achim Röll and Leif Bonorden (2022). "Improving API Design Skills with the API Design Fest: Insights from a Preliminary Experiment with Students". In: *24. Workshop Software-Reengineering und -Evolution (WSRE 2022)*. Vol. 42/2. Softwaretechnik-Trends, pp. 51–52 (cit. on pp. 145, 149).

Peter Saint-Andre, Dave Crocker, and Mark Nottingham (2012). *Deprecating the "X-" Prefix and Similar Constructs in Application Protocols*. RFC 6648. DOI: 10.17487/RFC6648 (cit. on p. 103).

Malavika Samak, Deokhwan Kim, and Martin C. Rinard (2020). "Synthesizing Replacement Classes". In: *Proceedings of the ACM on Programming Languages* 4.POPL. DOI: 10.1145/3371120 (cit. on p. 153).

Anand Ashok Sawant, Maurício Aniche, Arie van Deursen, and Alberto Bacchelli (2018a). "Understanding Developers' Needs on Deprecation as a Language Feature". In: *40th International Conference on Software Engineering (ICSE 2018)*, pp. 561–571. DOI: 10.1145/3180155.3180170 (cit. on pp. 18, 27, 35, 153).

Anand Ashok Sawant, Guangzhe Huang, Gabriel Vilen, Stefan Stojkovski, and Alberto Bacchelli (2018b). "Why are Features Deprecated? An Investigation Into the Motivation Behind Deprecation". In: *34th International Conference on Software Maintenance and Evolution (ICSME 2018)*, pp. 13–24. DOI: 10.1109/ICSME.2018.00011 (cit. on p. 153).

Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli (2018c). "On the reaction to deprecation of clients of 4 + 1 popular Java APIs and the JDK". In: *Empirical Software Engineering* 23, pp. 2158–2197. DOI: 10.1007/s10664-017-9554-9 (cit. on p. 153).

Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli (2019). “To react, or not to react: Patterns of reaction to API deprecation”. In: *Empirical Software Engineering* 24, pp. 3824–3870. DOI: 10.1007/s10664-019-09713-w (cit. on p. 153).

Simone Scalabrino, Gabriele Bavota, Mario Linares-Vasquez, Michele Lanza, and Rocco Oliveto (2019). “Data-Driven Solutions to Detect API Compatibility Issues in Android: An Empirical Study”. In: *16th International Conference on Mining Software Repositories (MSR 2019)*, pp. 288–298. DOI: 10.1109/MSR.2019.00055 (cit. on p. 154).

Simon Schneider and Riccardo Scandariato (2023). “Automatic extraction of security-rich dataflow diagrams for microservice applications written in Java”. In: *Journal of Systems and Software* 202, 111722. DOI: 10.1016/j.jss.2023.111722 (cit. on p. 64).

Souhaila Serbout and Cesare Pautasso (2024a). “APIstic: A Large Collection of OpenAPI Metrics”. In: *21st International Conference on Mining Software Repositories*, pp. 265–277. DOI: 10.1145/3643991.3644932 (cit. on pp. 78, 80).

– (2024b). “How Many Web APIs Evolve Following Semantic Versioning?” In: *24th International Conference on Web Engineering (ICWE 2024)*, pp. 344–359. DOI: 10.1007/978-3-031-62362-2_25 (cit. on pp. 20, 40, 80).

Snigdha Singh and Anne Koziolek (2024). “Automated Reverse Engineering for MoM-Based Microservices (ARE4MOM) Using Static Analysis”. In: *21st International Conference on Software Architecture (ICSA 2024)*, pp. 12–22. DOI: 10.1109/icsa59870.2024.00010 (cit. on p. 135).

Snigdha Singh, Dominik Werle, and Anne Koziolek (2022). “ARCHI4MOM: Using Tracing Information to Extract the Architecture of Microservice-Based Systems from Message-Oriented Middleware”. In: *16th European Conference on Software Architecture (ECSA 2022)*, pp. 189–204. DOI: 10.1007/978-3-031-16697-6_14 (cit. on p. 135).

César Soto-Valero, Thomas Durieux, and Benoit Baudry (2021a). “A Longitudinal Analysis of Bloated Java Dependencies”. In: *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. ESEC/FSE ’, pp. 1021–1031. DOI: 10.1145/3468264.3468589 (cit. on p. 52).

César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry (2021b). “A comprehensive study of bloated dependencies in the Maven ecosystem”. In: *Empirical Software Engineering* 26, 45. DOI: 10.1007/s10664-020-09914-8 (cit. on pp. 52, 61).

Sandro Speth, Elias Müller, Philipp Recke, Niklas Krieger, Steffen Becker, Alexander Poth, and Olsi Rjolli (2025). "Recovering Gropius Models with the Cluster Architecture Recovery Assistant". In: *22nd International Conference on Software Architecture (ICSA 2025)*. DOI: 10.1109/ICSA-C65153.2025.00025 (cit. on p. 52).

Kai Spichale (2025). *API-Design. Praxishandbuch für Java- und Webservice-Entwickler*. German. 3rd ed. dpunkt (cit. on p. 19).

S. Alexander Spoon (2007). "Fine-Grained API Evolution for Method Deprecation and Anti-Deprecation". In: *2007 International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)* (cit. on pp. 25, 154).

Klaas-Jan Stol and Brian Fitzgerald (July 2018). "The ABC of Software Engineering Research". In: *ACM Transactions on Software Engineering and Methodology* 27.3, 11, pp. 1–51. DOI: 10.1145/3241743 (cit. on p. 127).

– (2020). "Guidelines for Conducting Software Engineering Research". In: *Contemporary Empirical Methods in Software Engineering*. Ed. by Michael Felderer and Guilherme Horta Travassos, pp. 27–62. DOI: 10.1007/978-3-030-32489-6_2 (cit. on p. 29).

Margaret-Anne Storey, Neil A. Ernst, Courtney Williams, and Eirini Kalliamvakou (2020). "The who, what, how of software engineering research: a socio-technical framework". In: *Empirical Software Engineering* 25, pp. 4097–4129. DOI: 10.1007/s10664-020-09858-z (cit. on pp. 22, 28–30, 37, 127).

Roman Štrobl and Zdeněk Troníček (2013). "Migration from Deprecated API in Java". In: *4th Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH 2013) – Companion Proceedings*, pp. 85–86. DOI: 10.1145/2508075.2508093 (cit. on p. 154).

Matúš Sulír, Jaroslav Porubän, and Sergej Chodarev (2026). "Local software buildability across Java versions". In: *Empirical Software Engineering* 31, 78. DOI: 10.1007/s10664-026-10806-6 (cit. on p. 53).

Nabhan Suwanachote, Yagut Shakizada, Yutaro Kashiwa, Bin Lin, and Hajimu Iida (2025). "On the Evolution of Unused Dependencies in Java Project Releases: An Empirical Study". In: *22nd International Conference on Mining Software Repositories (MSR 2025)*, pp. 324–328. DOI: 10.1109/msr66628.2025.00059 (cit. on p. 52).

Richard N. Taylor, Nenad Medvidovič, and Eric M. Dashofy (2009). *Software Architecture: Foundations, Theory, and Practice*. Wiley (cit. on pp. 12–14).

Ferdian Thung, Stefanus A. Haryono, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang (2020). "Automated Deprecated-API Usage Update for Android Apps: How Far are We?" In: *27th International Conference on Software Analysis, Evolution and Reengineering (SANER 2010)*, pp. 602–611. DOI: 10.1109/SANER48275.2020.9054860 (cit. on p. 154).

Ferdian Thung, Hong Jin Kang, Lingxiao Jiang, and David Lo (2019). "Towards Generating Transformation Rules without Examples for Android API Replacement". In: *35th International Conference on Software Maintenance and Evolution (ICSME 2019)*, pp. 213–217. DOI: 10.1109/ICSME.2019.00032 (cit. on p. 154).

Jaroslav Tulach (2012). *Practical API Design. Confessions of a Java Framework Architect*. Apress (cit. on p. 19).

Joshi Unmesh (2024). *Patterns of Distributed Systems*. Addison-Wesley. 426 pp. (cit. on p. 16).

Aparna Vadlamani, Rishitha Kalicheti, and Sridhar Chimalakonda (2021). "APIScanner - Towards Automated Detection of Deprecated APIs in Python Libraries". In: *43rd International Conference on Software Engineering (ICSE 2021) – Companion Proceedings*, pp. 5–8. DOI: 10.1109/ICSE-Companion52605.2021.00022 (cit. on pp. 29, 154).

Vaughn Vernon (2013). *Implementing Domain-Driven Design*. Addison-Wesley (cit. on p. 19).

– (2016). *Domain-Driven Design Distilled*. Addison-Wesley (cit. on p. 19).

Melina Vidoni (2022). "A systematic process for Mining Software Repositories: Results from a systematic literature review". In: *Information and Software Technology* 144, 106791. DOI: 10.1016/j.infsof.2021.106791 (cit. on pp. 52, 63).

Jiawei Wang, Li Li, Kui Liu, and Haipeng Cai (2020a). "Exploring How Deprecated Python Library APIs Are (Not) Handled". In: *28th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, pp. 233–244. DOI: 10.1145/3368089.3409735 (cit. on p. 154).

Jiawei Wang, Li Li, and Andreas Zeller (2020b). "Better Code, Better Sharing: On the Need of Analyzing Jupyter Notebooks". In: *42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER 2020)*, pp. 53–56. DOI: 10.1145/3377816.3381724 (cit. on p. 154).

Roel J. Wieringa (2014). *Design Science Methodology for Information Systems and Software Engineering*. Springer (cit. on p. 43).

Erik Wilde (2019a). *Link Relation Types for Web Services*. RFC 8631. DOI: 10.17487/RFC8631 (cit. on p. 103).

– (2019b). *The Sunset HTTP Header Field*. RFC 8594. DOI: 10.17487/RFC8594 (cit. on pp. 18, 103).

Erik Wittern, Annie T.T. Ying, Yunhui Zheng, Jim A. Laredo, Julian Dolby, Christopher C. Young, and Aleksander A. Slominski (2017). "Opportunities in Software Engineering Research for Web API Consumption". In: *1st International Workshop on API Usage and Evolution (WAPI 2007)*, pp. 7–10. DOI: 10.1109/wapi.2017.1 (cit. on p. 15).

Claes Wohlin (2014). "Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering". In: *18th International Conference on Evaluation and Assessment in Software Engineering (EASE 2014)*, 38. DOI: 10.1145/2601248.2601268 (cit. on pp. 23, 25).

Yaoguo Xi, Liwei Shen, Yukun Gui, and Wenyun Zhao (2019). "Migrating Deprecated API to Documented Replacement: Patterns and Tool". In: *11th Asia-Pacific Symposium on Internetware*, pp. 1–10. DOI: 10.1145/3361242.3361246 (cit. on p. 154).

Yue Xiao, Dhilung Kirat, Douglas Lee Schales, Jiyong Jang, Luyi Xing, and Xiaojing Liao (2025). "JBomAudit: Assessing the Landscape, Compliance, and Security Implications of Java SBOMs". In: *32nd Network and Distributed System Security Symposium (NDSS 2025)*, pp. 1–20. DOI: 10.14722/ndss.2025.240322 (cit. on p. 52).

Bowen Xu, Le An, Ferdian Thung, Foutse Khomh, and David Lo (2020). "Why reinventing the wheels? An empirical study on library reuse and re-implementation". In: *Empirical Software Engineering* 25, pp. 755–789. DOI: 10.1007/s10664-019-09771-0 (cit. on p. 154).

Shinhyung Yang, David Georg Reichelt, Reiner Jung, Marcel Hansson, and Wilhelm Hasselbring (2025). "The Kieker Observability Framework Version 2". In: *16th ACM/SPEC International Conference on Performance Engineering (ICPE 2025)*, pp. 11–15. DOI: 10.1145/3680256.3721972 (cit. on p. 95).

Jerin Yasmin (2021). "RESTful API Deprecation: An Empirical Study on Deprecation Practicesand a Method for Detecting Deprecated API Usages in Client-side Web Applications". Available at http://hdl.handle.net/1974/29015. MA thesis. Queen's University (cit. on p. 42).

Jerin Yasmin, Yuan Tian, and Jinqiu Yang (2020). "A First Look at the Deprecation of RESTful APIs: An Empirical Study". In: *36th International Conference on Software Maintenance and Evolution (ICSME 2020)*, pp. 151–161. DOI: 10.1109/ICSME46990.2020.00024 (cit. on pp. 27, 35, 42, 43, 80, 82, 154).

Haruhiko Yoshioka, Sila Lertbanjongngam, Masayuki Inaba, Youmei Fan, Takashi Nakano, Kazumasa Shimari, Raula Gaikovina Kula, and Kenichi Matsumoto (2025). “Do Developers Depend on Deprecated Library Versions? A Mining Study of Log4j”. In: *22nd International Conference on Mining Software Repositories (MSR 2025)*, pp. 314–318. DOI: 10.1109/msr66628.2025.00057 (cit. on p. 52).

Ted Young and Austin Parker (2024). *Learning OpenTelemetry. Setting Up and Operating a Modern Observability System*. O’Reilly (cit. on p. 95).

Jing Zhou and Robert J. Walker (2016). “API Deprecation: A Retrospective Analysis and Detection Method for Code Examples on the Web”. In: *24th International Symposium on Foundations of Software Engineering (FSE 2016)*, pp. 266–277. DOI: 10.1145/2950290.2950298 (cit. on p. 154).

Marc Zimmermann, Leif Bonorden, and Oliver Kopp (2026). “Migrating HTTP Client Libraries in Java”. In: *28. Workshop Software-Reengineering und -Evolution (WSRE 2026)*. Vol. 46/2. Softwaretechnik-Trends (cit. on pp. 145, 149).

Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso (2023). *Patterns for API Design. Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley (cit. on pp. 15, 16, 19, 128).

Affidavit

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Sofern im Zuge der Erstellung der vorliegenden Dissertationsschrift generative Künstliche Intelligenz (gKI) basierte elektronische Hilfsmittel verwendet wurden, versichere ich, dass meine eigene Leistung im Vordergrund stand und dass eine vollständige Dokumentation aller verwendeten Hilfsmittel gemäß der Guten wissenschaftlichen Praxis vorliegt. Ich trage die Verantwortung für eventuell durch die gKI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate.

I hereby declare and affirm that this doctoral dissertation is my own work and that I have not used any aids and sources other than those indicated. If electronic resources based on generative artificial intelligence (gAI) were used in the course of writing this dissertation, I confirm that my own work was the main and value-adding contribution and that complete documentation of all resources used is available in accordance with good scientific practice. I am responsible for any erroneous or distorted content, incorrect references, violations of data protection and copyright law or plagiarism that may have been generated by the gAI.

Hamburg, May 28, 2026

Leif Bonorden

Use of Generative AI

This thesis does not feature any AI generated content, text or figures. Code and Data in the supplementary material have not been generated or modified by AI tools either.

The AI tool *Grammarly* was used to check spelling and grammar. The AI tool *Google Gemini 3 (Thinking)* was used to optimize the LaTeX and BibLaTeX configurations.