Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Can Proposed Service Interface Metrics Effectively Evaluate the Quality of RESTful APIs? A Repository Mining Study on API Evolution

Steffen Schneider

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Dr. Justus Bogner
Commenced:	April 19, 2021
Completed:	October 19, 2021

Abstract

RESTful web services and APIs are popular in industry and represent one commonly used way to expose functionality via a well-defined and technology-agnostic interface. While these APIs can be analyzed based on best practices or antipatterns, their interface quality can also be evaluated with metrics. Several of the interface metrics proposed in literature have been implemented in the RAMA approach based on API documentation. To empirically evaluate the effectiveness of the proposed interface metrics, we analyzed a large sample of publicly available APIs and compared the metric values to feasible ground truths for software quality. API descriptions were analyzed using the RAMA CLI and software quality metrics were collected using SonarQube and git. We used multiple linear regression models to examine the correlation between API metrics and software quality metrics. Furthermore, we studied the trend of the correlation over the evolution of a project. Our results suggest that some API metrics statistically significantly correlate with maintainability metrics. However, the regression models and the trend of the correlation indicate that as a project evolves, an increasing number of factors besides the API metrics influence the quality of the source code.

Kurzfassung

RESTful Web Services und APIs sind in der Industrie weit verbreitet und stellen eine häufig genutzte Möglichkeit dar, Funktionen über eine klar definierte und technologieunabhängige Schnittstelle bereitzustellen. Während diese APIs auf der Grundlage von Best Practices oder Antipatterns analysiert werden können, kann ihre Schnittstellenqualität auch mit Metriken bewertet werden. Mehrere der in der Literatur vorgeschlagenen Schnittstellenmetriken wurden im RAMA-Ansatz auf der Grundlage der API-Dokumentation implementiert. Um die Effektivität der vorgeschlagenen Schnittstellenmetriken empirisch zu bewerten, haben wir eine große Stichprobe öffentlich verfügbarer APIs analysiert und die Metrikwerte mit realisierbaren Grundwahrheiten für Softwarequalität verglichen. Die API-Beschreibungen wurden mit der RAMA CLI analysiert und die Softwarequalitätsmetriken wurden mit SonarQube und Git gesammelt. Mit Hilfe multipler linearer Regressionsmodelle untersuchten wir die Korrelation zwischen API-Metriken und Softwarequalitätsmetriken. Außerdem untersuchten wir den Trend der Korrelation über die Entwicklung eines Projekts. Unsere Ergebnisse deuten darauf hin, dass einige API-Metriken statistisch signifikant mit Metriken zur Wartbarkeit korrelieren. Die Regressionsmodelle und der Trend der Korrelation deuten jedoch darauf hin, dass mit der Entwicklung eines Projekts neben den API-Metriken eine zunehmende Anzahl von Faktoren die Qualität des Quellcodes beeinflussen.

Contents

List of Figures

List of Tables

Acronyms

API application programming interface. 15

APL Average Path Length. 23

APO Arguments per Operation. 23

BRC Biggest Root Coverage. 23

CLI command line interface. 23

CWE Common Weakness Enumeration. 22

DATEOAS Documentation As The Engine Of Application State. 19

DMR Distinct Message Ratio. 24

DW Data Weight. 24

HATEOAS Hypermedia as the Engine of Application State. 55

HTTP Hypertext Transfer Protocol. 18

IANA Internet Assigned Numbers Authority. 20

JSON JavaScript Object Notation. 18

LGPL GNU Lesser General Public License. 21

LOC lines of code. 22

LoC$_{msg}$ Lack of Message-Level Cohesion. 24

LP Longest Path. 24

MIME Multipurpose Internet Mail Extension. 18

MSR Mining Software Repositories. 29

NCLOC non-commented lines of code. 30

NOR Number of Roots. 24

OWASP Open Web Application Security Project. 22

RAMA RESTful API Metric Analyzer. 15

RAML RESTful API Modeling Language. 21

REST Representational State Transfer. 15

RPC Remote Procedure Calls. 18

SANS	SysAdmin, Audit, Network, and Security. 22
SDK	software development kit. 20
SIDC	Service Interface Data Cohesion. 24
SLA	Service Level Agreement. 25
SOAP	Simple Object Access Protocol. 20
URI	Uniform Resource Identifier. 18
VIF	variance inflation factor. 35
WADL	Web Application Description Language. 20
WSDL	Web Services Description Language. 20
WSIC	Weighted Service Interface Count. 25
XML	Extensible Markup Language. 18
YAML	yet another markup language. 21

1 Introduction

RESTful web services and application programming interfaces (APIs) have seen an exponential increase in adoption in the last years and are the de-facto standard for synchronous communication [BFWZ19] [SCL16]. They are used by big companies such as Google, Amazon, Twitter, and Facebook to provide simple access to some of their resources for third-parties. The services following Representational State Transfer (REST) architectural principles expose functionality via a well-defined and technology-agnostic interface. The exposed interfaces are used by applications, leading to an increasing number of applications that depend on REST APIs to communicate. This poses several challenges for developers such as badly designed or documented APIs [Rob09]. Complex and difficult-to-use APIs increase development effort and thus costs. Therefore, high API quality and easy-to-use APIs should be aimed for.

While REST APIs can be analyzed based on best practices or antipatterns, their interface quality can also be evaluated with metrics. Several of the metrics proposed in literature have been implemented in the RESTful API Metric Analyzer (RAMA) approach, which is based on API documentation [Bog20]. However, the effectiveness of these metrics has not been empirically evaluated, i.e. we cannot be sure whether these metrics are valid proxies for software quality. Since the documentation and implementation of a lot of RESTful APIs are publicly available, a pragmatic approach to empirically evaluate the effectiveness of the proposed service interface metrics is to analyze the available APIs and compare the metric values to feasible ground truths for software quality. The results provide guidance whether the implemented metrics can predict, and hence serve as a valid proxy, for software quality.

The remainder of this work is structured as follows. Chapter 2 describes relevant technical background information, such as RESTful APIs and REST API description languages. Chapter 3 presents work related to our study. Chapter 4 contains the research questions, hypotheses, and a description of methods used in our study. In Chapter 5 we present the results, i.e. collected data, linear regression models, and the trend of API and software quality. Chapter 6 discusses the results and implications of our study, and presents threats to validity. Chapter 7 concludes our work and suggests future research.

2 Technical Background

This chapter contains background information about technologies and software related to our study.

2.1 ISO/IEC 25010

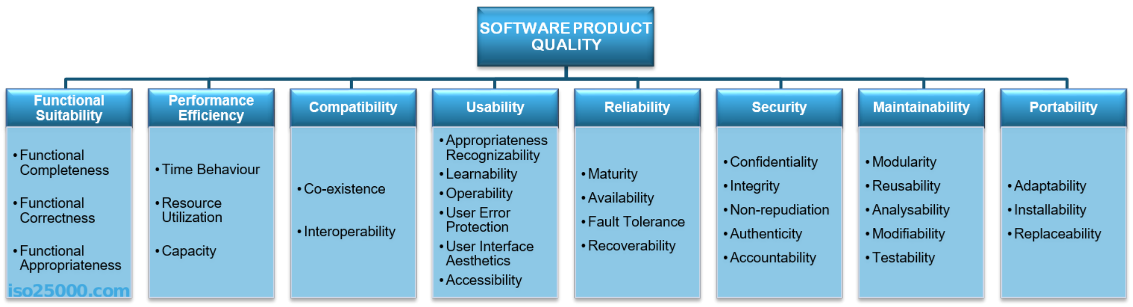


Figure 2.1: Software Product Quality characteristics defined in ISO/IEC 20510

ISO/IEC 25010 is a software quality standard published in 2011 [ISO11]. It is composed of eight categories that are used to define and evaluate software product quality (Figure 2.1).

- *Functional Suitability* represents the degree to which a product provides specified functions when used under specified conditions. Our study focuses on the sub-characteristic Functional Correctness, which is the degree to which a product provides the correct results. Other sub-characteristics are Functional Completeness and Functional Appropriateness.
- *Performance Efficiency* represents the capability of a product to provide appropriate performance relative to the amount of resources used under stated conditions. Sub-characteristics are Time Behaviour, Resource Utilization, and Capacity.
- *Compatibility* represents the degree to which a product can exchange information with other products and/or perform its required functions while sharing the same hardware or software environment. The sub-characteristics of Compatibility are Co-existence and Interoperability.
- *Usability* represents the degree to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction. The sub-characteristics are Appropriateness Recognizability, Learnability, Operability, User Error Protection, User Interface Aesthetics, and Accessibility.

- *Reliability* represents the degree to which a product performs specified functions under specified conditions for a specified period of time. The sub-characteristics are Maturity, Availability, Fault Tolerance, and Recoverability.
- *Security* represents the degree to which a product protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization. Sub-characteristics are Confidentiality, Integrity, Non-repudiation, Accountability, and Authenticity.
- *Maintainability* represents the degree of effectiveness and efficiency with which a system can be improved, corrected or adapted to changes in environment or requirements. Its sub-characteristics are Modularity, Reusability, Analysability, Modifiability, and Testability.
- *Portability* represents the degree of effectiveness and efficiency with which a system can be transferred from one environment to another. The three sub-characteristics are Adaptability, Installability, and Replaceability.

In our study, we focus on the quality (sub-) characteristics functional correctness, maintainability, and security.

2.2 RESTful APIs

A RESTful API is an API that follows the popular REST architectural style. It provides client-server communication based on Hypertext Transfer Protocol (HTTP), Uniform Resource Identifiers (URIs) and Multipurpose Internet Mail Extension (MIME) types. REST focuses on simplicity, scalability, and standards-based interoperability [WPR10]. Data is mostly exchanged in JavaScript Object Notation (JSON) or Extensible Markup Language (XML) format [PNIR20]. Other mechanisms are Remote Procedure Calls (RPC) and Hybrid approaches [MPD10].

RPC APIs expose internal functionality of a service through a complex programming-language-like interface. The remote procedure is called as if it were a local procedure. There are many different implementations of this concept that are not compatible [MPD10].

Hybrid approaches are a mix between RESTful and RPC. They use HTTP methods but implement different semantics, i.e. the behavior is contradictory to the used HTTP method. For example, *getNews* is realized via POST and addNews via GET. The use of hybrid APIs can be problematic since they do not guarantee operation safety. Data can be unintentionally manipulated, e.g., when crawlers use the GET method, expecting to retrieve information [MPD10].

2.2.1 Best practices

The use of best practices is very important for the design of APIs, as they have an impact on the understandability for third parties. Several papers have identified best practices and guidelines to follow when designing RESTful APIs. Giessler et al. describe eight categories of best practices [GGS+15].

No versioning of the API. Altough Mulloy states that a version identifier should be mandatory [Mul12], Giessler et al. conclude that versioning is not necessary due to hypermedia.

The **description of resources** should follow these five best practices:

- Use nouns for resource names.
- Use domain specific resource names.
- Limit the amount of resources to reduce complexity.
- Do not mix plural and singular names for resources.
- Consider naming conventions of JavaScript.

The **identification of resources** with URIs should follow these four best practices:

- The URI should be self-explanatory.
- Address a resource by a maximum of two URIs. One is for a collection, the other one for a specific element in the collection.
- The identifier of an element should be difficult to predict.
- Do not use verbs in the URI.

Error messages should be clear and understandable and realized using these best practices:

- Limit the amount of used HTTP methods.
- Use HTTP methods according to their official specification [NMM+99].
- Provide 1.) an error code, 2.) a message for developers, 3.) a message for users, and 4.) a hyperlink for more information in an error returned from the server.

Fielding suggests the **documentation of the Web API** should be prevented, because what method to use on what URIs should be defined by the media type [Fie08]. Giessler et al. however suggest to use their newly developed kind of documentation called Documentation As The Engine Of Application State (DATEOAS), which consists of 1) examples, 2) a state diagram to represent the relationship and state transition between resources, and 3) a reference book of all error codes for developers.

For the **usage of parameters**, which can extend an URI to forward optional information, Giessler et al. focus on four different widely used use cases.

- For filtering, either attributes of the information to filter or a special query language should be used.
- For sorting, they encourage the use of a comma separated list.
- The selection, i.e. the reduction of transmission size over the network by specifying the requested information, should be implemented with a URI parameter *fields* and a comma separated list of attributes.
- Pagination is used to split information on virtual pages with references to the next, previous, first, and last page. Recommended URI parameters are *offset* and *limit*. Offset defines the virtual page and limit the amount of information on the virtual page.

The **interaction with resources** should follow the three identified best practices:

- The HTTP methods should conform to the method's semantics as defined in the official HTTP specification.
- The OPTIONS method used to request the supported methods of a resource is recommended if a large amount of data has to be transmitted. It is only necessary if the supported operations are not specified in the representation.
- Conditional GET prevents the server from repeatedly sending the same information multiple times. The information should only be sent if it was modified since the last request.

For the **support of MIME types**, Giessler et al. identified four best practices. MIME types are used to identify data formats. They are registered and published by the Internet Assigned Numbers Authority (IANA).

- At least two data representation formats should be supported, e.g., JSON and XML.
- JSON should be the default representation format due to its wide-ranging adaptation.
- Existing MIME types should be used.
- Content negotiation to allow the client to choose the representation format should be supported via the HTTP header field *ACCEPT*.

2.3 Common REST API Description Languages

API description languages are domain-specific languages used to describe APIs in human- and machine-readable languages [Bie15]. They describe different properties of a RESTful API such as endpoints, schemas, and message-transfer-types. Since they are machine-readable, programs can generate client software development kits (SDKs) to interact with an external API by parsing its API description file. Additionally, HTTP handlers that implement an API description file can be generated automatically. We describe three REST API description languages used in our study.

2.3.1 WADL

Web Application Description Language (WADL) is an XML based language published in 2009 by Sun Microsystems [Sun09]. WADL is seen as the REST equivalent of Web Services Description Language (WSDL), which is used to describe Simple Object Access Protocol (SOAP)-based web services. Although WSDL2.0 can describe REST services, WADL is more lightweight and easier to understand [Gos20]. However, it is not standardized and considered complicated in practice due to its XML syntax.

2.3.2 RAML

RESTful API Modeling Language (RAML) is a yet another markup language (YAML) based language to specify "practically" RESTful APIs, i.e. APIs that do not follow all principles of REST. RAML's goal is to design new APIs, not just describe existing ones [Sch16]. It was proposed in 2013 with support from technology leaders such as MuleSoft, AngularJS, PayPal, and Cisco.

Mulesoft joined the OpenAPI Initiative in 2017, indicating that the industry is converging on a single dialect to describe APIs, namely OpenAPI. Future versions of RAML might build on top of the OpenAPI specification [Tam17].

2.3.3 OpenAPI

OpenAPI, previously known as Swagger, is a description language for RESTful APIs. Swagger was created in 2011 by Tony Tam, donated to the OpenAPI Initiative created by the Linux Foundation in 2015, and renamed to OpenAPI Specification (OAS) in 2016. The specification can be written in JSON or YAML format, since both are convertible into each other [KK18]. It was first designed to document APIs, but evolved to manage the whole lifecycle of APIs [Sch16].

The terms Swagger and OpenAPI should not be used interchangeably. OpenAPI refers to the specification, while Swagger refers to the tool suite around OpenAPI. Examples are the Swagger-UI [1] to interact with APIs via user interface, or OpenAPI Generator [2] to generate client SDKs or server stubs.

2.4 SonarQube

SonarQube is an open-source platform to collect code quality metrics via static analysis [Son21e]. It is released under the GNU Lesser General Public License (LGPL) [Fre07] and written in Java. The stable version used in this study is 8.9 (May 4, 2021).

SonarQube supports metrics regarding complexity, code duplication, maintainability, reliability, security, size, and tests. The metrics used in our study are as follows:

- *Cognitive Complexity*: Cognitive Complexity represents the level of difficulty for developers to understand the code's control flow. The score is based on three basic rules [Cam21]:
  - Shorthand structures (e.g., a?.myObj) are ignored.
  - Increment the score for each break in the linear flow of the code (e.g., loops and conditionals).
  - Increment the score for nested flow-break structures.

[1] https://swagger.io/tools/swagger-ui/
[2] https://openapi-generator.tech/

- *Duplicates lines density*: The density of duplicated lines is the percentage of lines involved in code duplication. It is calculated by

$$\text{Duplicated lines density} = \frac{\text{Duplicated lines}}{\text{Lines of code} * 100}$$

  Literature refers to the metric as Clone Coverage [OW14].

- *Code Smells*: A code smell is a maintainability issue. The number of code smells includes issues such as too many duplicated lines of code or high function complexity.
- *Functions*: Functions represents the total number of functions.
- *Files*: Files represents the total number of files.
- *Code-to-Comment ratio*: The Code-to-Comment ratio (Comments %) represents the density of comment lines and is computed by

$$\text{Code-to-Comment Ratio} = \frac{\text{Comment lines}}{\text{Lines of code + Comment lines}}$$

  A value of 50 % represents an equal distribution of lines of code and comment lines, 100 % means there are only commented lines.

- *Vulnerabilities*: Vulnerabilities are problems that impact the application's security and need to be fixed immediately. SonarQube uses different representations of the source code under the hood to detect any kind of security issue [Son21d]. The rules to detect security issues are 1.) *security-injection rules*, such as SQL Injection (CWE-89), Cross-Site Scripting (CWE-79), or Code Injection (CWE-94), and 2.) *security-configuration rules*, such as Sensitive Cookie Without 'HttpOnly' Flag (CWE-1004), Improper Validation of Certificate with Host Mismatch (CWE-297), or Use of a Broken or Risky Cryptographic Algorithm (CWE-327). Security rules cover standards, such as Common Weakness Enumeration (CWE) [The21], Open Web Application Security Project (OWASP) [OWA17], and SysAdmin, Audit, Network, and Security (SANS) [SAN11] (outdated). A vulnerability in the SonarQube report typically comprises of one or more lines of code (LOC), that belong together semantically. SonarQube's target is to have more than 80 % of true-positives vulnerabilities [Son21c].

3 Related Work

In this chapter, related work that is relevant for our study is presented and discussed.

3.1 RESTful API Metric Analyzer (RAMA)

RAMA is an approach for the static analysis of RESTful APIs proposed by Bogner [Bog20]. The approach to evaluate the quality of RESTful APIs is implemented in a prototypical tool, the RAMA command line interface (CLI). The quality is measured by several structural metrics. Seven of them are complexity metrics, two are cohesion metrics, and one is a size metric.

Before describing RAMA metrics, we need to define some terms, such as resource, operation and parameter.

> *Resources* are endpoints (paths), e.g. "/users/1" [Sma21].
>
> *Operations* in RAMA are the combination of a resource and an HTTP verb or method (GET, POST, PUT, PATCH, or DELETE), e.g., "GET /users" [Bog20].
>
> *Parameters* are variables that are replaced by concrete values. Different types of parameters are path parameters ("/user/{id}"), query parameters ("/users?role=admin"), header parameter ("X-MyHeader: Value"), and cookie parameter ("Cookie: debug=0") [Sma21].

The metrics implemented by RAMA are

- *Arguments per Operation (APO)*: This complexity metric represents the average amount of operation arguments for a service. It is calculated by dividing the summed up number of arguments for all service operations by the total number of operations. Arguments are path parameters and request bodies [BM09]. Lower values are considered better, according to a benchmark test conducted by Bogner et al. [BWZ20].
- *Average Path Length (APL)*: This complexity metric represents the average length of the resource paths. The path length of a single resource is defined by the number of slashes ("/") in the resource. Slashes at the end of the path are ignored. The average path length is computed by summing up the path length of each resource and dividing it by the total number of resource paths [HLSV17]. A lower APL is better.
- *Biggest Root Coverage (BRC)*: This complexity metric is defined as the percentage of operations located under the largest root path element. The root path element is the string between the first and second slash ("/"). BRC has a value range between 0 and 1, with larger values considered better. It is calculated by dividing the number of operations from the root resource with most operations by the total number of operations [HLSV17].

- *Data Weight (DW)*: This complexity metric represents the complexity of data types of input and output messages. It is computed by counting all path parameters and all parameters in request and response bodies within all operations [BM09]. Lower DW values are better.
- *Distinct Message Ratio (DMR)*: This complexity metric represents the complexity/cohesion of data types in an interface. It is computed by measuring the ratio between distinct messages and all messages inside an interface [BM11]. Lower values of DMR are better. RAMA implements a significantly modified version of the original metric [Bog20].
- *Lack of Message-Level Cohesion (LoC$_{msg}$)*: This cohesion metric represents a measure for the cohesion of an interface [AZM+15]. Lower values are better since they indicate less lack of cohesion. Message-Level Cohesion assumes that two operations are related if their input (respectively, output) data are similar. Input data is defined as the combination of path parameters, query parameters, and request body. Output data is defined as the combination of all responses. The similarity of input data of two operations and output data of two operations is calculated by:

  $$\text{inputDataSimilarity} = \frac{\text{commonInputDataProperties}}{\text{unionOfAllInputDataProperties}}$$

  $$\text{outputDataSimilarity} = \frac{\text{commonOutDataProperties}}{\text{unionOfAllOutDataProperties}}$$

  The similarity between operations is then calculated by:

  $$\text{operationSimilarity} = (\text{inputDataSimilarity} + \text{outputDataSimilarity})$$

  For each pair of operations, the following formula is used to calculate the LoC$_{msg}$ [ESE19]. The *numberOfPairs* is the number of all possible pairs, thus it is calculated with the binomial coefficient:

  $$\text{numberOfPairs} = \binom{\text{totalOperationCount}}{2}$$

  LoC$_{msg}$ is calculated by:

  $$\text{LoC}_{msg} = \frac{\sum_{i=0}^{\text{numberOfPairs}} 1 - \text{operationSimilarity}_i}{\text{numberOfPairs}}$$

- *Longest Path (LP)*: This complexity metric represents the longest path of a resource in an interface. Equally to the APL, the path length is defined as the number of slashes ("/") in the resource. Slashes at the end of the path are ignored [HLSV17]. A lower value is better.
- *Number of Roots (NOR)*: This complexity metric is defined as the number of root resources with a distinct root path element [HLSV17]. Less distinct roots are considered better.
- *Service Interface Data Cohesion (SIDC)*: This cohesion metric measures the cohesion of a service (S) based on the cohesiveness of the operations ($SO(SI_S)$) of its exposed interface (SI_S). Operations are deemed to be cohesive if they share the same input parameter or return types. A service is deemed to be cohesive if all possible pairs of operations have at least one common parameter and return type. The values for SIDC range from 0 to 1, with value 1

representing the strongest possible cohesion and 0 indicating total lack of cohesion [PRF07]. Formally defined,

$$SIDC(S) = \frac{\text{Common}(\text{Params}(SO(SI_S))) + \text{Common}(\text{returnTypes}(SO(SI_S)))}{\text{Total}(SO(SI_S)) \cdot 2}$$

where

Common (Params ($SO(SI_S)$)) is the function that returns the number of service operation pairs that share at least one input type.

Common (returnTypes ($SO(SI_S)$)) is the function that returns the number of service operation pairs that share the same return type.

Total ($SO(SI_S)$) is the function that returns the number of all possible combinations of operation pairs for the service interface SI_S. It is calculated by:

$$Total(SO(SI_S)) = \frac{(n-1) \cdot n}{2}$$

with n denoting the number of operations in the interface.

- *Weighted Service Interface Count (WSIC)*: This size metric represents the weighted number of exposed operations in an interface. The default weight is 1. Other weighting methods, which need to be empirically validated, can be based on number and complexity of data types of parameters in the interface. By default, WSIC returns the number of exposed methods [HCA09], where lower values are considered better.

  Hirzalla et al. observed that a higher number of interfaces per service lead to an increasing complexity, due to

  - an increasing amount of work required to specify, construct and test every interface,
  - an increasing amount of monitoring required to ensure that Service Level Agreements (SLAs) are met, and
  - performance and problem determination concerns becoming primary issues with an increase in complexity of individual interfaces of data structures for a given service [HCA09].

Examples for the calculation of metrics can be found in the documentation of RAMA on GitHub [ESE19].

The APIs to analyze are provided as machine-readable API documentation. RAMA supports the RESTful API description languages OpenAPI V3 [Lin21], RAML 1.0 [RAM16], and WADL [Sun09]. The RAMA tool is realized as CLI application. It reads an API description file and reports the resulting metrics as JSON or PDF file. Its architecture is loosely based on the *pipes and filters* style [LS04]. The tool was developed at the at the Software Engineering Group [1] of the University of Stuttgart. It is available as open source project on GitHub [ESE19].

[1] iste.uni-stuttgart.de/ese/

3.2 API Evolution

APIs change during their lifecycle to include new features, remove ambiguities, or fix bugs. Di Lauro et al. examined the evolution of open source Web APIs [DSP21]. They observed that most APIs change multiple times within a day while other changes occur after the API was untouched for more than three years. Often, the API description files are committed once and then left untouched. The change frequency of APIs does not diminish with growing age. The size, measured as the number of paths, of 50 % of examined APIs increase over time, while 6 % decrease. 44 % do not change in size.

Eilertsen and Bagge classify changes by how they affect the client [EB18]. Changes can be made in terms of 1.) metadata (e.g., renaming a library), 2.) syntax (e.g., changing a method signature), and 3.) semantics. One challenge when evolving APIs is to handle breaking changes. Breaking changes lead to client code running erroneously until updated and adapted to the API changes. Breaking semantic changes are the most difficult to detect.

3.3 Software Quality and its Relation to API Quality

The ISO/IEC 25010 standard defines software quality as “the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value” [ISO11]. There are several quality models that provide characteristics and measurements for software quality, e.g. ISO 25000 [ISO14] and Quamoco [WGH+15]. The ISO 25000 is a series of standards that revise the earlier ISO 9126 [ISO01]. It is comprised of four parts: Quality model (ISO 25010), Quality measurement (ISO 25020), Quality requirements (ISO 25030), and Quality evaluation (ISO 25040). ISO standards are widely referred to in research, indicating their importance [NNR19]. Quamoco bridges the gap between abstract quality aspects and concrete quality measurements by operationalized quality models [WGH+15].

Tahmooresi et al. observed that using complicated external APIs results in more defects in the code [THN20]. The complexity of external APIs was measured using crowd-related metrics, such as the number of discussions on StackOverflow [2]. Kim et al. studied the impact of API-level refactorings on software quality and the software development process [KCK11]. The results indicate that the time taken to fix bugs decreases by about 35.0 % after refactorings. However, the rate of bugfixes also increases after API-level refactorings. This might be due to bugs being introduced during refactorings or the fact that refactorings help developers identify latent bugs. Alrubaye showed that library API migrations improve software quality in terms of reduced coupling, increased cohesion, and improved code readability [Alr19].

Compared to the aforementioned studies, we examine RESTful APIs, not source-code level APIs. Moreover, we focus on code implementing an API, not code that uses an external API. Code that implements an API is probably written by the same developers that designed the API, whereas external APIs are designed by different developers. Therefore, results of our study might not coincide with existing studies.

[2] `https://stackoverflow.com/`

4 Study Design

In this chapter, we describe the design of our study. We present four research questions and eight hypotheses. Furthermore, we describe selection criteria of study objects, data collection procedures, operationalization of software quality, as well as analysis and validity procedures. Last, we characterize our study objects.

4.1 Research Questions

This study focuses on whether metrics implemented by RAMA can predict software quality characteristics. Additionally, we examine how the prediction changes over the evolution of a project.

RQ1: Can metrics implemented by RAMA predict maintainability?

The metrics implemented by RAMA cover the maintainability of services [Bog20]. Therefore, our first research questions focuses on the maintainability of services implementing RESTful APIs. We formulate six hypotheses focusing on aspects of software maintainability such as complexity and size.

RQ2: Can metrics implemented by RAMA predict functional correctness?

Lack of cohesion is not only correlated with software maintainability, but also functional correctness [KEE12]. Since two metrics implemented by RAMA measure cohesion (LoC$_{msg}$, SIDC), we investigate whether API metrics can predict the functional correctness of a service. We formulate one hypothesis related to the prediction of functional correctness by API metrics.

RQ3: Can metrics implemented by RAMA predict security?

Medeiros et al. describe that a lack of cohesion in functions has a strong positive linear correlation with the number of vulnerabilities [MICV17]. We examine whether the API quality is also correlated with the number of vulnerabilities by formulating one hypothesis regarding software security.

RQ4: How does the prediction evolve over time?

The last research questions exploratively examines how the predictions of RQ1 - RQ3 change over the course of the evolution of projects. The result helps determine whether API metrics can predict software quality in the early stages of a project or whether projects need to reach a certain level of maturity.

4.1.1 Hypotheses

We formulate eight hypotheses about RAMA metrics and software quality before performing data collection and analysis. H_1 - H_6 focus on software maintainability, H_7 on functional correctness, and H_8 on software security.

#	**Hypotheses on RAMA metrics and software quality**
H_1	Projects with poor RAMA metrics have more complex code
H_2	Projects with poor RAMA metrics have more code duplication
H_3	Projects with poor RAMA metrics have more Code Smells
H_4	Projects with poor RAMA metrics have longer functions
H_5	Projects with poor RAMA metrics have longer files
H_6	Projects with poor RAMA metrics have lower Code-to-Comment ratio
H_7	Projects with poor RAMA metrics have more bugs
H_8	Projects with poor RAMA metrics have more vulnerabilities

Table 4.1: Overview of hypotheses

The reasoning for H_1 is that poor RAMA metrics indicate higher complexity in the API, which might lead to higher complexity in the implementation of the API. Poor RAMA metrics, especially low data cohesion, might lead to code being distributed across multiple packages/files. This could lead to more code duplication (H_2). Code smells (H_3) are maintainability-related issues in the code [Son21b], such as unused variables, empty functions, and unnecessary semicolons. Projects with lower API quality might show an overall lower maintainability, indicated by a higher number of code smells. H_4 and H_5 are based on the assumption that APIs with high complexity and many parameters lead to more LOC since all parameters have to be validated and processed. This might lead to many LOC per file and function. The reasoning for H_6 is that experienced developers which design better APIs might leave more comments in their code. The motivation behind H_7 is that low cohesion and high complexity in the API lead to developers not implementing certain edge-cases and conditions, hence introducing bugs into the system. Medeiros et al. observed that coupling and complexity show a very strong correlation with the number of vulnerabilities [MICV17]. Therefore, poor RAMA metrics might lead to more vulnerabilities being introduced into a system.

We reject a hypothesis if there are no API metrics that correlate with software quality. The operationalization of software quality is described in Section 4.5.

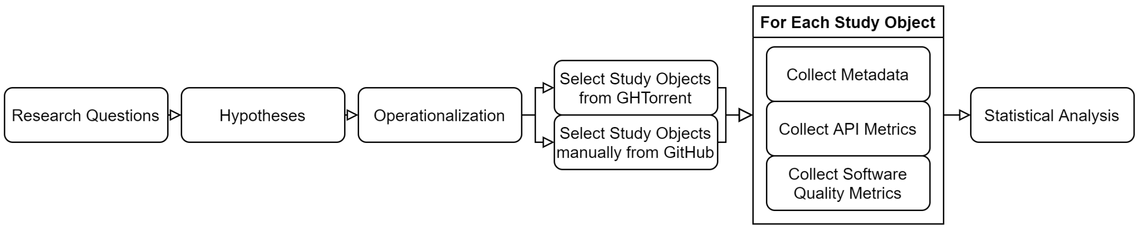


Figure 4.1: Overview of the research process

4.2 Overview of the Research Process

This Mining Software Repositories (MSR) study was conducted in multiple steps. Figure 4.1 provides an overview of the research process. First, we derived research questions to evaluate the effectiveness of proposed service interface metrics. Second, we developed hypotheses to answer the research questions. Most hypotheses focus on RQ1, since most of the proposed service interface metrics are maintainability-related. Third, we reviewed literature to identify reputable and sound metrics to operationalize the quality aspects of the hypotheses. Fourth, we defined criteria for study objects to include and exclude. Fifth, we collected data according to the criteria previously defined. Suitable study objects were identified via the GHTorrent dataset and a manual search on GitHub. We collected API metrics using the RAMA CLI and software quality metrics using SonarQube and git. Sixth, we constructed a linear model to statistically analyze the data.

4.3 Study Object Selection

We used GitHub [1] to find projects containing machine-readable API documentation and source code implementing the API. GitHub provides a REST API to retrieve metadata and raw data from projects [Git21]. The API has a rate limit of 5,000 authenticated requests per hour.

The GHTorrent dataset [Gou13] offers query capabilities for metadata of GitHub projects in the form of mongoDB or MySQL database dumps or a Google BigQuery database [2]. The fileserver of the TU Delft [3] provided the newest version (March 2021) in the form of a MySQL dump, hence we used this dataset.

Suitable candidates for static analysis were selected by 1) filtering the GHTorrent dataset and 2) manually searching GitHub. First, we compiled a preselection of suitable candidates by querying the GHTorrent dataset. Since the GHTorrent dataset did not allow to query for repositories containing YAML or RAML files, we used the GitHub API to filter the resulting dataset which contained more than 700,000 projects. We searched repositories for API description files, i.e. RAML, WADL, OpenAPI and Swagger files. RAML and WADL files can be identified by their respective file ending .raml and .wadl. To identify OpenAPI and Swagger files, we parsed the file content of JSON-

[1] https://github.com/
[2] https://ghtorrent.org/gcloud.html
[3] http://ghtorrent-downloads.ewi.tudelft.nl/mysql/

and YAML-files for typically occurring strings such as “swagger:” or “openapi:”. We verified the resulting set of repositories manually to confirm the exclusion of duplicates and projects for test or learning purposes (see Section 4.3.1).

Additionally, we used the Advanced Search on GitHub [4] to search for repositories with API description files. This step was conducted manually since the search API does not allow to search for code inside files globally across GitHub. The search queries used are

- “extension:yml extension:yaml extension:json openapi”,
- “extension:yml extension:yaml extension:json swagger”,
- “extension:yml extension:yaml extension:json filename:openapi”,
- “extension:yml extension:yaml extension:json filename:swagger”,
- “extension:yml extension:yaml extension:json filename:apidescription”,
- “extension:yml extension:yaml extension:json filename:api”,
- “extension:yml extension:yaml extension:json filename:service”,
- “extension:wadl extension:raml”.

Again, we used the same filter criteria as described in Section 4.3.1 to exclude unsuitable repositories.

4.3.1 Inclusion and exclusion criteria for source code repositories

The following list describes inclusion and exclusion criteria for repositories.

- *Required files*: The project must use OpenAPI V3, Swagger, RAML, or WADL in order to be analyzable by the RAMA CLI. Swagger and RAML 0.8 files were converted to OpenAPIV3 and RAML 1.0. Additionally, it must contain source code implementing the API description files.
- *Languages*: The programming language of the repository is supported by SonarQube [Son21a], i.e. C, C++, GoLang, Java, JavaScript, Kotlin, PHP, Python, Ruby, Scala, Typescript.
- *Minimum size of code base*: The repository must have more than 200 non-commented lines of code (NCLOC) after excluding non-relevant files. Small repositories that are only dummy repositories could distort the results.
- *Minimum size of API*: The API of the repository must have at least three exposed operations (WSIC >= 3), one root (NOR >= 1), and a path length of at least 1 (LP >= 1). Repositories with small or empty APIs could distort the results.
- *No Fork or Mirror*: The repository must not be a fork or a mirror of another repository. This is applied to avoid analyzing the same code base multiple times.

[4] `https://github.com/search/advanced`

- *Maximum Clone Coverage*: The Clone Coverage of the repository must be below 75 %. This value is based on a study conducted by Roehm et al. [RVWJ19]. Repositories with a value above 75 % often contain multiple copies of the same project.
- *No test/learning/examples projects*: The repository must not be 1.) for testing purposes of another repository, 2.) for learning purposes, and 3.) examples to illustrate coding books. They are small repositories which might bias the results. These types of repositories were identified by a keyword search in repository name and description. Keyword examples are “example”, “petstore”, or “tutorial”.

4.4 Data Collection Procedures

After selecting suitable repositories, we cloned them and excluded irrelevant files that might distort the results of the analysis.

For RQ4, we collected data for 50 version of a repository. The versions are equally distributed across the history of git commits, e.g. if the repository has 1000 commits, the 20^{th}, 40^{th}, 60^{th}, ... commit is considered one version. This enables the comparison of repository versions on a uniform scale. We analyzed the largest repositories w.r.t. NCLOC of our sample because a large code-base indicates that the repository has evolved and reached a certain level of maturity.

The collection of data for one repository (version) is as follows:

1. *Clone and checkout a repository (version)*: A repository was cloned using git and the correct git version was checked out. For RQ1 - RQ3 we checked out the latest version of the main branch. For RQ4 we checked out the selected version.
2. *Retrieve metadata*: For the latest version, we retrieved metadata such as the number of stars and forks using the GitHub API. [Git21].
3. *Exclusion of irrelevant files*: We excluded API description files and source code that might distort the results. API description files that are not implemented by the repositories’ source code impact the results of the analysis with RAMA. This includes API description files for testing purposes and source code of libraries that were included in the repository. Examples for API description files used for testing purposes are projects that generate API description files such as the OpenAPI Generator project [5].

   For the collection of software quality metrics, we excluded generated, test, frontend, and library code. Code implementing API descriptions can be generated by tools, such as the OpenAPI Generator project. Generated code leads to more code duplication, since the generated code is usually not read by humans and is constructed in a way that is considered a code clone. We exclude test code because its quality varies between repositories and is not always up-to-date with the source code [SD15]. Frontend code is typically JavaScript code that runs in the browser. Since it does not implement the API of the backend service, it is excluded. Many repositories contain library code, i.e. code developed by a third party. Since

[5] https://openapi-generator.tech/

this type of code is often compiled, minified, and not part of the primary code implementing API description files, we also exclude library code (cf. Roehm et al. [RVWJ19], Steidl and Deissenboeck [SD15]).

Irrelevant code was detected using the following techniques:

- **Word exclusion**: Irrelevant API description files are detected using strings such as “Swagger Sample API”. The list includes three strings.

  Words that indicate that the source code file is part of the frontend, library, generated, or test code, are for example “React”, “<html>”, or “JUnit”. This list includes 54 words.
- **Path exclusion**: Paths that indicate the directory includes irrelevant code and API files are for example “examples”, “dist”, or “test”. This list includes 242 paths.
- **File name exclusion**: We created a list of (patterns of) file names of irrelevant files, such as popular third party libraries, generated code, and code for testing or demonstration purposes. Examples are “**/d3*.js” or “**/*.spec.*”. This list includes 105 (patterns of) file names.
- **File type exclusion**: We excluded files with file-endings that indicate that the file contains source code that does not implement an API, e.g. “.css” or “.vue”. This list includes 18 file types.

All lists for exclusions were compiled and verified manually using SonarQube’s and RAMA’s result reports. SonarQube’s report visualizes the quality of the analyzed source code per file and directory. RAMA’s report is a JSON file with metadata of the analyzed API description file and the calculated metric values. The list of exclusion criteria is provided in the digital appendix [Sch21].

4.5 Operationalization of Software Quality

For our study, we focus on metrics related to the software quality aspects maintainability, functional correctness, and security (see Section 2.1). We chose the following metrics because they have been used in previous studies ([Aba21] [RVWJ19]), they are mostly language-independent, and they are considered to be suitable to evaluate software quality ([OW14] [FS15]). Furthermore, they are statically analyzable and therefore suitable for a large-scale study.

There are two common types of source code complexity. Cyclomatic Complexity and Cognitive Complexity [Son21b]. Cyclomatic Complexity measures the complexity by representing a program as a control flow graph and counting its independent execution paths. This metric by McCabe is criticized because it does not measure the complexity of a program as perceived by a human [OW14]. Cognitive Complexity is considered a better alternative to Cyclomatic Complexity and it is empirically validated to be able to reflect some aspects of code understandability [BWW20]. Among other criteria, it is based on the nesting depth of code, which is considered a good metric to measure maintainability [OW14]. The Cognitive Complexity mentioned in Table 4.2 is the sum of the complexity of all files. Since large programs have more code and therefore a higher total complexity, we normalize the Cognitive Complexity by the number of NCLOC. A lower Cognitive Complexity indicates better maintainability. We use Cognitive Complexity to evaluate H_1.

Hypothesis	**Metric**	**Definition**
H_1	Cognitive Complexity / #NCLOC	The comprehensibility of the code's control flow [Son21b] The complexity value is normalized by NCLOC
H_2	Clone Coverage (%)	Ratio of lines of code involved in code duplication
H_3	Code Smells / #NCLOC	The number of maintainability-related issues [Son21b] per NCLOC
H_4	#NCLOC / #Functions	Mean #NCLOC per function
H_5	#NCLOC / #Files	Mean #NCLOC per file
H_6	Code-to-Comment ratio (%)	Ratio of lines of code to lines of comments [Son21b]
H_7	Bugfixing Commits (%)	Ratio of commits to fix a bug in the git commit history
H_8	#Vulnerabilities / #NCLOC	Number of vulnerability issues per NCLOC [Son21b]

Table 4.2: Overview of static analysis metrics used to analyze maintainability, functional correctness, and security

Code duplication is problematic because if an error is corrected in one part of the code, it still persists in the code clone. The density of code duplication is also known as "Clone Coverage" [OW14] and used to evaluate H_2. A lower value indicates better maintainability.

To evaluate H_3, we calculate the mean number of code smells per line of code. Code smells are poor implementation and design decisions that affect the maintainability of a program negatively [TJL17]. Code smells also include functions with Cognitive Complexity greater than 15 and duplicated code. Less code smells indicate better maintainability.

H_4 and H_5 are evaluated by volumetric maintainability metrics. We calculate the mean lines of code per function (files) by dividing the NCLOC of a repository by the total number of functions (files). Long functions and files impact maintainability negatively because developers that change code have to consider much code.

Code-to-Comment Ratio represents the density of comment lines and is considered a good metric to measure software maintainability [OW14]. Code that is commented well is easier to understand for someone that needs to change the code. The metric used for H_6 is represented as percentage value. A value of 50 % indicates that the number of lines of code is equal to the number of lines of comments. 100 % means that there are only comment lines.

For H_7, we adapted the metric of counting bugfixing commits used by Abajirov to measure functional correctness of functional programming languages [Aba21]. It is represented as percentage value and calculated by

$$\text{Bugfixing Commits (\%)} = \frac{\text{\#Bugfixing Commits}}{\text{\#Total Commits}} \times 100$$

A lower value indicates better functional correctness due to the absence of bugs that needed to be fixed.

Vulnerabilities are security-related issues reported by CWE [The21]. Large repositories with more NCLOC potentially have more vulnerabilities, since more NCLOC offer a larger attack vector. Therefore, we normalize the number of vulnerabilities by the number of NCLOC. Less vulnerabilities per NCLOC indicate better security.

4.6 Analysis Procedures

We performed static analysis to calculate metric values, descriptive analysis for an overview of the data, and inferential statistics with multiple linear regression to test our hypotheses. We used R to perform statistical analysis.

4.6.1 Static Analysis

RAMA Analysis: We use the RAMA CLI tool to analyze API description files in a repository. RAMA metrics are described in Section 3.1.

Since RAMA only supports OpenAPI V3, RAML 1.0 and WADL, we convert Swagger (OpenAPI V2) and RAML 0.8 files. We use the oas-kit [6] to convert Swagger files to OpenAPI V3 and the oas-raml-converter [7] to convert RAML 0.8 to RAML 1.0. RAMA computes API metrics for a single API description file and provides a report in JSON. Since numerous repositories contain multiple API description files, e.g., one for authentication and one for other functionality, we aggregate RAMA metric values from all JSON report files. Manual inspection indicates that some API description files are duplicates, i.e. they describe different versions of the same service or are part of compiled files while the other one is part of source files. Therefore, if there are multiple API description files with the same file name or identical metric values, we select the report of the first analyzed file for aggregation.

The aggregation process differed between metrics. For APO, APL, DMR, LoC$_{msg}$, and SIDC we calculate the arithmetic mean. For BRC and LP, we choose the maximum metric value. For NOR, we parse the list of root paths provided in the JSON reports and count the distinct root paths. For example, RAMA report of API description file A provided root paths “/api, /user”, and RAMA report of API description file B provided root paths “/api, /group”. The number of distinct root paths is three (/api, /user, /group). Simply summing up could have distorted the aggregated values when there are duplicates of API description files. For DW and WSIC, we calculate the sum of the individual RAMA results. In contrast to NOR, there were no additional information provided by RAMA to prevent summing up duplicates.

Gather Ground Truth: We use SonarQube [Son21b] and git to gather static software quality metrics. SonarQube metrics are described in Section 2.4.

For H_1, we use SonarQube’s metric “Cognitive Complexity”. All SonarQube metrics are retrieved via the SonarQube server’s RESTful API. For H_2, we use SonarQube’s “code duplication density” metric as a measure for the percentage of lines of code that are involved in duplicated code. A

[6] https://github.com/Mermade/oas-kit

[7] https://github.com/mulesoft/oas-raml-converter

duplication is 100 successive and duplicated tokens spread across ten lines of code for Non-Java projects, and ten successive and duplicated statements regardless of tokens and lines for Java projects [Son21b]. Indentation and string literal differences are ignored. For H_3, we consider all code smells provided by SonarQube. For H_4 (H_5), we use the number of NCLOC and number of functions (files) provided by SonarQube to compute the mean. For H_6, we use the Code-to-Comment ratio provided by SonarQube and multiply it by 100 to present it as percentage value. For the ratio of Bugfixing Commits (H_7), we consider all commits on the main branch of a repository, including merge and revert commits. Bugfixing Commits are identified by a keyword search in the commit message and commit body. We use a list of 68 keywords assembled by Abajirov [Aba21]. For H_8, we use the number of vulnerabilities reported by SonarQube and normalize it by the number of NCLOC.

4.6.2 Statistical Analysis

Since no data attribute is normally distributed, we report median instead of mean to describe groups of repositories and metrics. For all hypotheses we compare all RAMA metrics to a single ground truth, i.e. a software quality metric. Therefore, we use multiple linear regression with the *lm* package in R [8] as the method of analysis. The multiple linear regression model has the form of

$$\begin{aligned}\mathbf{DV} = \beta_0 + \beta_1\mathbf{APO} + \beta_2\mathbf{APL} + \beta_3\mathbf{BRC} + \beta_4\mathbf{DMR} + \beta_5\mathbf{DW} + \beta_6\mathbf{LoC}_{msg} + \\ \beta_7\mathbf{LP} + \beta_8\mathbf{NOR} + \beta_9\mathbf{SIDC} + \beta_{10}\mathbf{WSIC} + \epsilon\end{aligned} \tag{4.1}$$

where DV denotes the dependent variable (e.g. Cognitive Complexity), β_0 the constant y-intercept, β_i the slope coefficient and ϵ the error term of the model. The API metrics are explanatory (independent) variables.

We examined the sample data for multicollinearity, heteroskedasticity, multivariate normality, and auto-correlation. For all hypotheses, the variance inflation factor (VIF) indicated strong multicollinearity between SIDC and BRC ($VIF_{SIDC} > 19$, $VIF_{BRC} > 16$). Hence, we excluded SIDC from all regression models. We use the Breusch-Pagan test to test for heteroskedasticity [BP79]. Heteroskedasticity describes that variances of residuals in a linear model are not equal. For all hypotheses, we cannot reject the null hypotheses of the Breusch-Pagan test, concluding there is no heteroskedasticity. We use Q-Q plots and the Shapiro-Wilk test to examine the presence of multivariate normality, i.e. that residuals are normally distributed. Non-normality can be rectified by removing outliers, transforming the data, fitting other distributions, or using non-parametric tests. If the Q-Q plot shows right-skewed data [Yea21], we transform the data by applying the square root to the dependent variables of the regression model [Cho21]. The square root transformation can be applied because we only have positive values in our dataset. Since we do not want to fit a perfect model but understand the relationship between independent and dependent variable, we tolerate that the Shapiro-Wilk test for normality on the residuals after transformation reports a p-value less than 0.05. We use the Durbin-Watson test to check for autocorrelation. Autocorrelation is the degree of correlation between values of the same variable across different observations. The null hypothesis of the test is that residuals are not autocorrelated. For all our hypotheses, we could not reject the

[8] https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/lm

null hypothesis. To counteract the problem of multiple comparisons, we apply the Holm-Bonferroni correction [Hol79]. This way to deal with familywise error rates (FWER) is more powerful than the Bonferroni correction [AG96].

If a multiple linear regression model indicates an independent variable correlates with the dependent variable, we perform Pearson's correlation test for the independent variable (RAMA metric) and dependent variable (software quality metric) to report the correlation coefficient r.

For RQ4, we calculated Pearson's correlation coefficient r for every RAMA metric and software quality metric for each of the 50 versions. Pearson's correlation evaluates the correlation of a linear relationship of two continuous variables. The values of the API metrics and software quality metrics are continuous. We removed observations without pairs of values, i.e. that were missing either the API metric or the software quality metric.

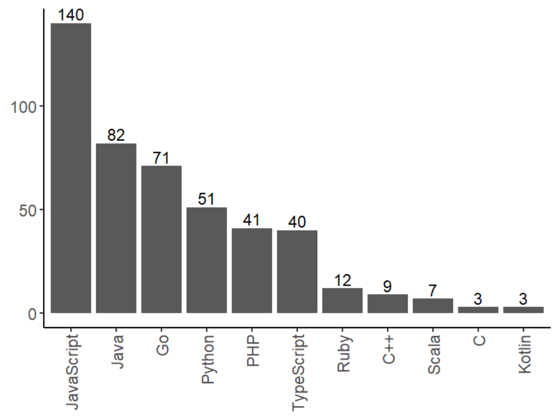
4.7 Validity Procedures

During data collection, we took several measures to maximize the validity of the study results. First, we excluded forks of repositories and manually verified the absence of duplicates. Repositories with similar names and identical folder structure were manually excluded, since they are probably duplicates although they are not marked as forks by GitHub. Second, we manually verified the results of more than 50 RAMA and SonarQube analyses. This was performed to improve the filter criteria for files and directories to exclude. Third, we used procedures that are resistant to duplicates in API description files to prevent a distortion of values when aggregating RAMA results of multiple API description files. Examples are counting the distinct roots programmatically instead of summing up the provided values for the metric NOR. Fourth, we use metrics that are viable for all repositories. For example, some GitHub features such as issues or release count are only used by a fraction of repositories [KGB+14]. Hence we refrain from using them. Fifth, all data exclusion and analysis was realized in an automatic manner, removing the threat of human variance and error. This also improves the replicability of the study. Sixth, we use more than 450 repositories for testing all hypotheses of RQ1 - RQ3. Lastly, we use a conservative significance level for our statistical tests ($\alpha = 0.01$).

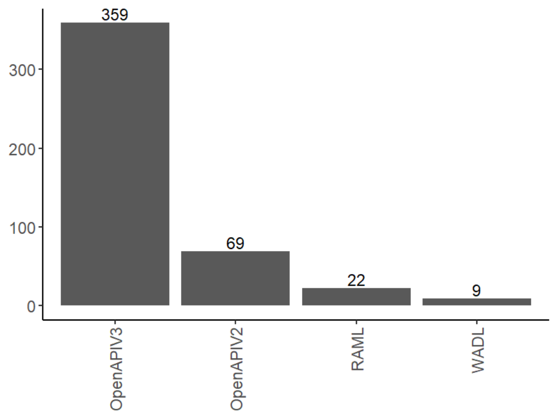
4.8 Study Objects

Figure 4.2a presents the distribution of primary languages as provided by the GitHub API [Git21]. Most repositories have JavaScript as primary language (140), followed by Java (82) and Go (71). Kotlin (3) and C (3) are the primary language of the fewest repositories. As shown in Figure 4.2b, the most common API description type is OpenAPI V3 (359 repositories), followed by its predecessor Swagger (OpenAPI V2) with 69 repositories. 22 repositories implement RAML files and 9 implement WADL files.

Figure 4.3 provides an overview of the metadata of the sample containing 459 repositories. Only repositories that fulfill all criteria as described in Section 4.3.1 are included in the sample. The size is the required disk space of the git repository, i.e. before excluding irrelevant files such as frontend or library files. The median repository size is 15.28 MB with values ranging from 0.08 MB to



(a) Language distribution



(b) Distribution of API description types

Figure 4.2: Distribution of languages and API description types

1,415.25 MB. The NCLOC is provided by SonarQube after irrelevant files are excluded. The median NCLOC is 4,081 with values ranging from 206 to 560,052. C repositories have the highest median value of 27,675 NCLOC, followed by Ruby repositories with 18,111. JavaScript and TypeScript repositories have the lowest median value with 1,796 and 2,337 NCLOC respectively. These values could be explained by the unbalanced language distribution in our sample. The #commits includes all commits on the main branch. The median #commits is 229 with values ranging from 1 to 506,792. Ruby repositories have the highest median value of 3,072,5 commits, followed by C with 2811 commits. Again, JavaScript (114.5) and TypeScript (175) repositories have the lowest median value. The #stars and the #forks is provided by the GitHub API [Git21]. Most repositories in our sample have 0 stars (183 repositories) and 0 forks (202 repositories). The median #stars is 1 with values ranging from 0 to 39,072. The median #forks is 1 with a value range from 0 to 5,674.

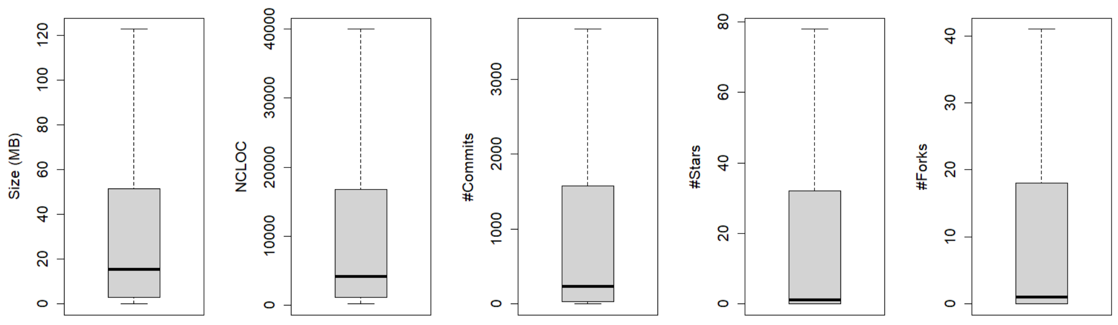


Figure 4.3: Overview of the study objects

5 Results

This chapter describes the metrics of the collected data, the results of the statistical analysis using multiple linear regression, and the results of the analysis of the evolution over time.

5.1 General Descriptive Statistics

Metric	**Median**	**Minimum**	**Maximum**
APO	1.33	0	16
APL	2	0.422	6.294
BRC	0.667	0.022	1
DW	105	0	25,084
DMR	0.349	0	1
LoC_{msg}	0.783	0	1
LP	4	1	18
NOR	4	1	76
SIDC	0.554	0.022	1
WSIC	17	3	1,430
Cognitive Complexity	0.134	0	0.739
Clone Coverage (%)	2.1	0	71.1
Code Smells	0.021	0	0.528
Function length	13.556	3.315	1,700.986
File length	63.455	2.195	1,455.06
Code-to-Comment ratio (%)	12.8	0	86.6
Bugfixing commits (%)	12.245	0	78.947
Vulnerabilities	0	0	3.88×10^{-03}

Table 5.1: Descriptive statistics of metric results

Table 5.1 shows descriptive statistics of the collected metrics. The table contains the name of the metric, as well as median, minimum, and maximum value calculated from the sample of 501 repositories. All values of metrics presented in the results are normalized, e.g. “Cognitive Complexity” represents “Cognitive Complexity / #NCLOC” (see Table 4.2).

RAMA: The median value of APO is 1.33 with values ranging from 0 to 16. The median path length is 2 with a minimum of 0.422 and maximum of 6.294. The median value of BRC is 0.667 with the lowest value of BRC of 0.022 and the some repositories exposing their operations under only one root (BRC = 1). There is a large spike in the distribution of values of BRC at 1 and a smaller spike between 0.3 and 0.5, indicating most APIs have all operations nested under one root or equally distributed among 2 or 3 roots. SIDC has a similar value distribution as BRC. DW has a median of 105 and maximum of 25,084. The minimum value of WSIC is 3 due to the filter criteria defined in Section 4.3.1.

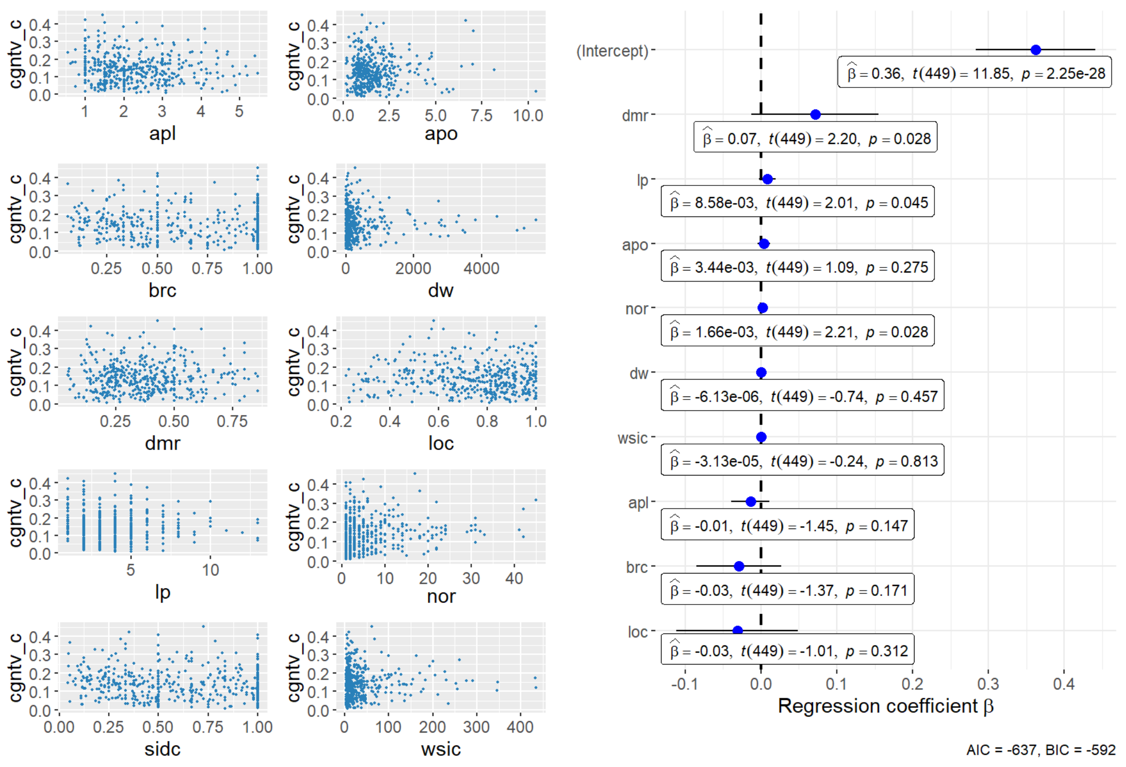
Ground Truths: The median Cognitive Complexity is 0.134 with a minimum of 0 and maximum of 0.739. In the median, 2.1 % of lines are duplicated with values ranging from 0 % to 71.1 %. The median number of Code Smells per NCLOC is 0.021. The maximum value is 0.528 Code Smells per NCLOC. The median function length is 13.556 with a minimum of 3.315 and maximum of 1,700.986 NCLOC per function. The median file length is 64.455 with values ranging from 2.195 to 1,455.06. 12.8 % of lines are comments with a minimum value of 0 %. The project with the highest Code-to-Comment ratio has 86.6 % comments. The same repository also has the most Code Smells. 12.245 % of commits are bugfixing commits, with a maximum value of over 78 %. Most repositories have no vulnerabilities, as indicated by a median value of 0. The repository with the most vulnerabilities has 3.88×10^{-03} vulnerabilities per NCLOC.

Comparison of the sample API metric values to a benchmark: Bogner et al. performed a benchmark test to classify which values for the respective metrics implemented by RAMA can be considered as good or bad [BWZ20]. The thresholds of metric values are divided in quartiles, each representing 25 % of their analysed API description files. The quartiles are “top 25 %”, “25 % - 50 %”, “50 % - 75 %”, and “worst 25 %” of projects. Each quartile has a range of values that indicates the quality of the metric, e.g. the top 25 % of projects have an APO value of 0.20 - 3.52. Table 5.1 shows the median values of our sample. The median values of our sample for APO, and APL are in top 25 % of metric values according to the benchmark by Bogner et al. The median values of DMR, DW, and LP are in the top 25 % - 50 %. The median values of BRC, SIDC, and WSIC are in the worst 50 % - 75 %. The median values of LoC_{msg} and NOR are in the worst 25 % of values of the benchmark. The values for APO and DW indicate that our sample tends to contains smaller APIs. Although small, LoC_{msg} and SIDC suggest that the APIs have low cohesion compared to the benchmark sample.

5.2 RQ1: Can metrics implemented by RAMA predict maintainability?

In this section we present the results related to maintainability aspects of software quality (RQ1).

5.2.1 H_1 Projects with poor RAMA metrics have more complex code



(a) Distribution of RAMA metrics and Cognitive Complexity **(b)** Correlation of RAMA metrics and Cognitive Complexity

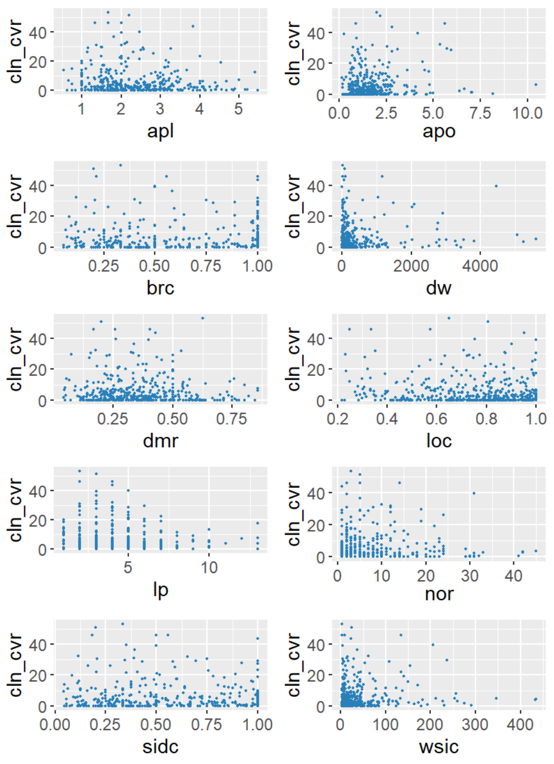
Figure 5.1: Distribution (a) and correlation (b) of RAMA metrics and Cognitive Complexity

Figure 5.1a shows the distribution of data of RAMA metrics and Cognitive Complexity. Each subplot displays data points of a singular RAMA metric on the x-axis and Cognitive Complexity on the y-axis. For all plots showing the data distribution in this chapter we removed outliers (top and bottom one percent of data w.r.t. the software quality metric and RAMA metric) for better visualization. The plots emphasize the similarities between the distribution of BRC and SIDC, and DW and WSIC. Furthermore, the concentration of values of BRC and SIDC at 1.0 and 0.5 is visible. Figure 5.1b shows the correlation of RAMA metrics and Cognitive Complexity. We applied the square root function to all data points before fitting the model to correct for right-skewness and non-normality of residuals. Shapiro Wilk's normality test for the residuals reports a p-value of 0.04141 after transformation. The model does not show a statistically significant correlation between an API metric and Cognitive Complexity. Since we chose a significance level of 0.01, neither DMR (p = 0.028), LP (p = 0.045), nor NOR (p = 0.028) is considered statistically significantly correlated.

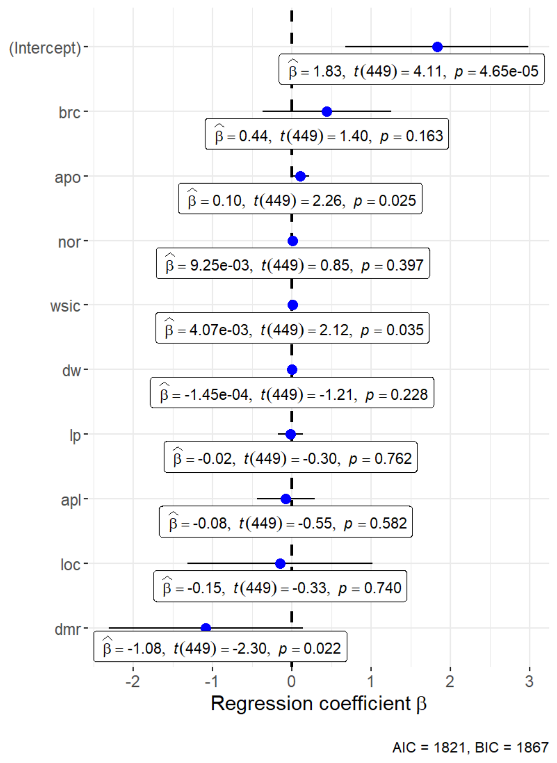
WSIC (p = 0.813) and DW (p = 0.457) are least significantly correlated with Cognitive Complexity. The model has a p-value of less than 2×10^{-16}, but is only able to explain less than four percent of variability in the data (adjusted $R^2 = 0.0343$).

Since there are no RAMA metrics that correlate with Cognitive Complexity, we reject H_1.

5.2.2 H_2 Projects with poor RAMA metrics have more code duplication



(a) Distribution of RAMA metrics and Clone Coverage



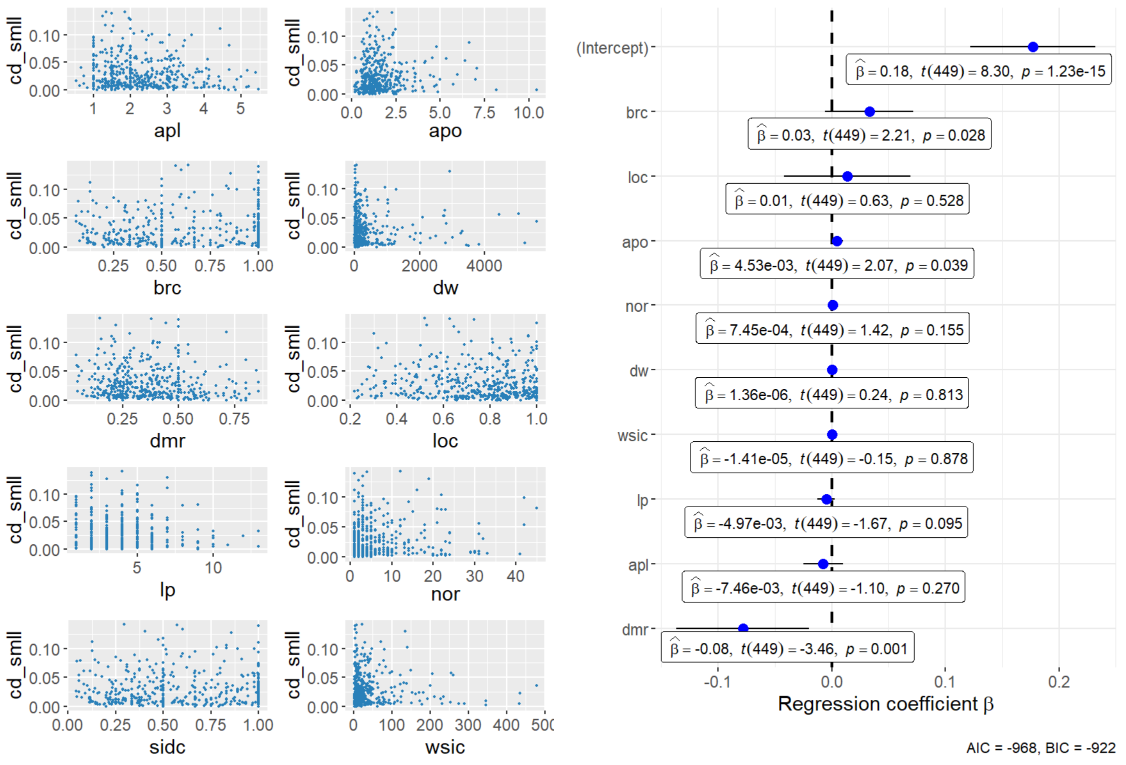
(b) Correlation of RAMA metrics and Clone Coverage

Figure 5.2: Distribution (a) and correlation (b) of RAMA metrics and Clone Coverage

Figure 5.2a shows the distribution of data of RAMA metrics and Clone Coverage. Most values of Clone Coverage are smaller than 20, with some outliers with larger values. Figure 5.2b shows the correlation of RAMA metrics and Clone Coverage. After transforming the data by applying the square function to the dependent variable, the residuals are not normally distributed (Shapiro-Wilk's p-value < 2.2×10^{-16}), which we accept as described in Section 4.6.2. No RAMA metric correlates statistically significantly with Clone Coverage ($\alpha = 0.01$). APO ($\beta = 0.10$, p = 0.025), WSIC ($\beta = 4.07 \times 10^{-03}$, p = 0.035) are the most significantly positively correlated metrics, while DMR ($\beta = -1.08$, p = 0.022) is the most significantly negatively correlated metric. The model has a p-value of 4.65×10^{-05}, but is only able to explain less than four percent of variability in the data (adjusted $R^2 = 0.0361$).

No RAMA metric is statistically significantly correlated with Clone Coverage. Therefore, we reject H_2.

5.2.3 H_3 Projects with poor RAMA metrics have more Code Smells



(a) Distribution of RAMA metrics and Code Smells **(b)** Correlation of RAMA metrics and Code Smells

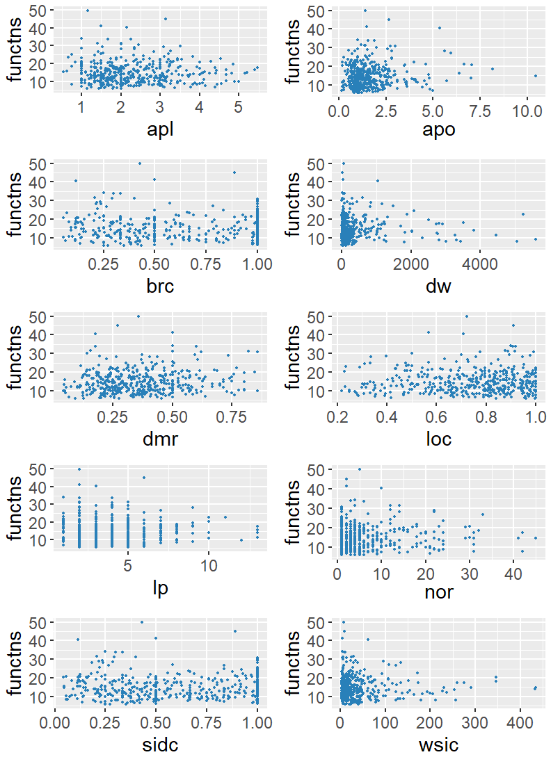
Figure 5.3: Distribution (a) and correlation (b) of RAMA metrics and Code Smells

Figure 5.3a shows the distribution of Code Smells. Figure 5.3b shows the correlation of RAMA metrics and Code Smells. We transformed the dependent variable (Code Smells) using the square-root function. However, the residuals still show non-normality (Shapiro Wilk test's p-value = 9.53×10^{-13}). The model shows a statistically significant negative correlation between DMR and the number of Code Smells ($\beta = -0.08$, p = 0.001). Pearson's correlation coefficient for DMR and Code Smells indicates a small effect size (r = -0.142, p = 0.002283) [Coh88].

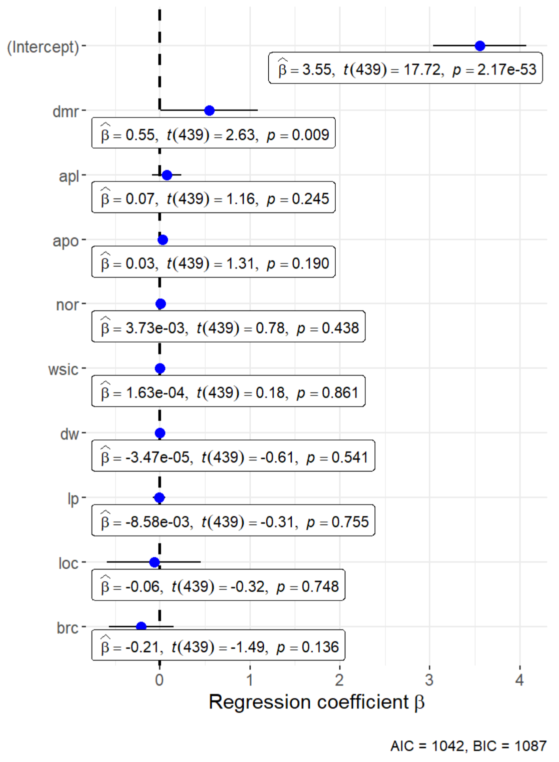
Other interesting positive correlations are APO ($\beta = 4.53 \times 10^{-03}$, p = 0.039) and BRC ($\beta = 0.03$, p = 0.028). The model has a p-value of 1.23×10^{-15}, but is only able to explain less than five percent of variability (adjusted $R^2 = 0.04778$).

DMR correlates statistically significantly with the number of Code Smells. Therefore, we cannot reject H_3.

5.2.4 H_4 Projects with poor RAMA metrics have longer functions



(a) Distribution of RAMA metrics and average function length

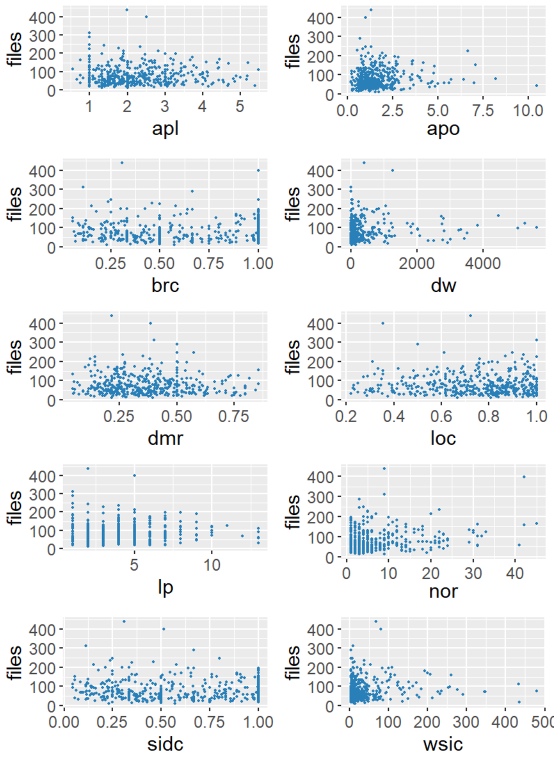


(b) Correlation of RAMA metrics and average function length

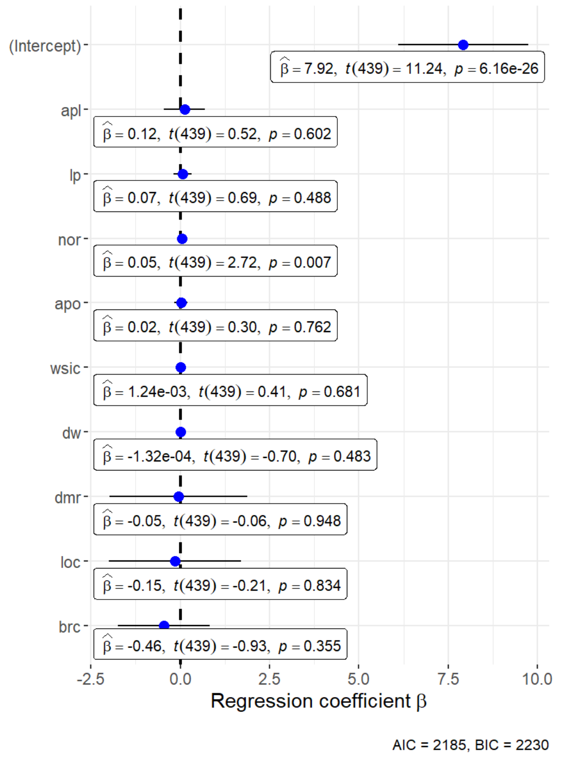
Figure 5.4: Distribution (a) and correlation (b) of RAMA metrics and average function length

Figure 5.4a shows the data distribution of RAMA metrics and average function length. Figure 5.4b shows the correlation of RAMA metrics and average function length. We removed outliers (top and bottom 1 % of of data w.r.t. average function length) that distorted the Q-Q plot before applying the square root function to the dependent variable. Shapiro-Wilk's normality test for the distribution of residuals has a p-value of 4.724×10^{-09} after transforming the data. The model shows a statistically significant positive correlation between DMR and the average function length ($\beta = 0.55$, p = 0.009). Pearson's correlation coefficient for DMR and the average function length is 0.111 with a p-value of 0.01862, indicating a small effect size. The model has a p-value of less than 2×10^{-16}, but can only explain about one percent of variability in the data (adjusted $R^2 = 0.009613$).

DMR is statistically significantly positively correlated with the average function length. Therefore, we cannot reject H_4.



(a) Distribution of RAMA metrics and average file length



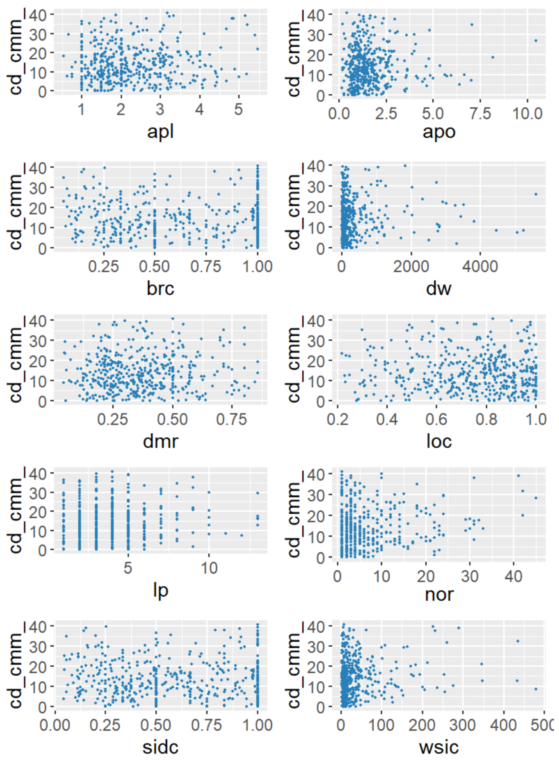
(b) Correlation of RAMA metrics and average file length

Figure 5.5: Distribution (a) and correlation (b) of RAMA metrics and average file length

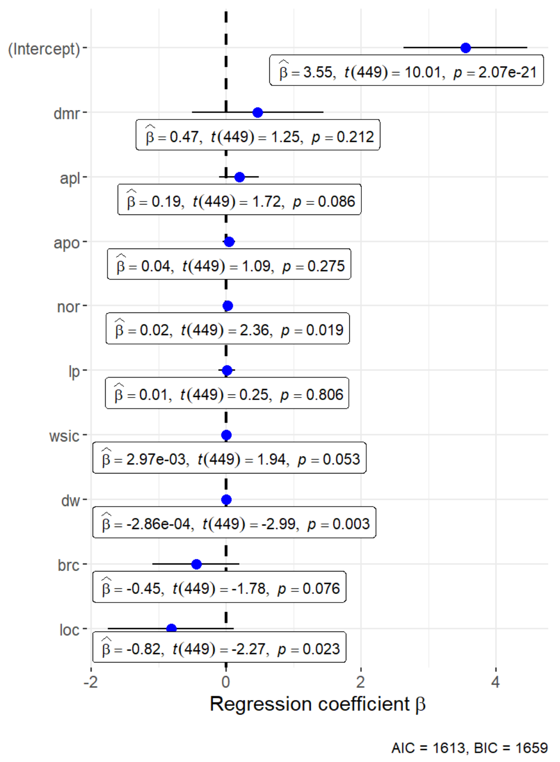
5.2.5 H_5 Projects with poor RAMA metrics have longer files

Figure 5.5a shows the distribution of RAMA metrics and average file length. Figure 5.5b shows the correlation of RAMA metrics and average file length. Again, we removed outliers and applied the square root function to the dependent variable due to right-skewed residuals. Shapiro-Wilk's normality test reports a p-value of 2.15×10^{-10} for the residuals after transformation, hence the residuals are not normally distributed. The API metric NOR statistically significantly correlates with the average file length ($\beta = 0.05$, p = 0.007). The effect size of the correlation is small to medium (Pearson's r = 0.1715, p = 0.0002606). The model has a p-value of less than 2×10^{-16}, but can only explain two percent of variability (adjusted $R^2 = 0.02028$).

Since the NOR is statistically significantly positively correlated with the average file length, we cannot reject H_5.



(a) Distribution of RAMA metrics and Code-to-Comment ratio



(b) Correlation of RAMA metrics and Code-to-Comment ratio

Figure 5.6: Distribution (a) and correlation (b) of RAMA metrics and Code-to-Comment ratio

5.2.6 H_6 Projects with poor RAMA metrics have lower Code-to-Comment ratio

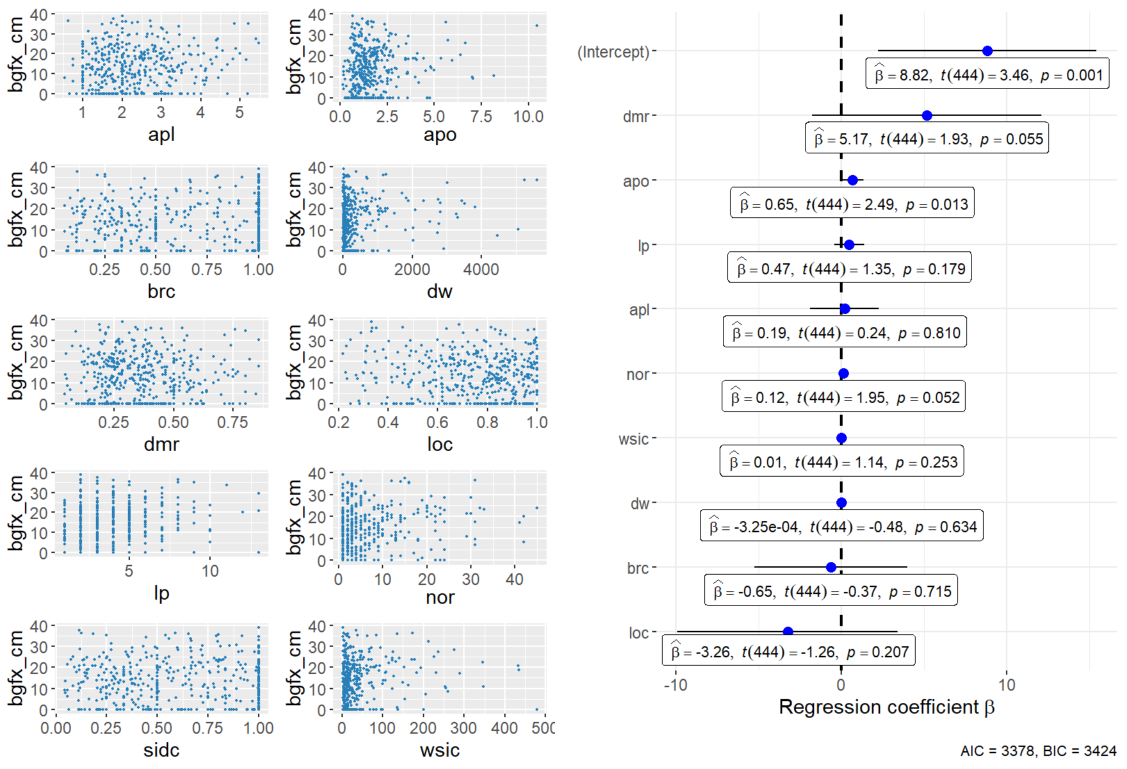
Figure 5.6a shows the distribution of RAMA metrics and Code-to-Comment ratio. Figure 5.6b shows the correlation of RAMA metrics and Code-to-Comment ratio. Due to the right-skewed distribution of residuals, we applied the square root function to the dependent variable. Subsequently, Shapiro-Wilk's normality test indicates normally distributed residuals (p-value = 0.08424). We identified a statistically significant negative correlation between DW ($\beta = -2.86 \times 10^{-04}$, p = 0.003) and Code-to-Comment ratio. Pearson's correlation coefficient r of DW and Code-to-Comment ratio indicates a small effect size (r = -0.0428, p-value = 0.3602). Other interesting correlations are LoC$_{msg}$ and NOR. LoC$_{msg}$ and Code-to-Comment ratio have a negative correlation ($\beta = -0.82$, p = 0.023), NOR and Code-to-Comment ratio a positive ($\beta = 0.02$, p = 0.019). The model has a p-value of less than 2×10^{-16}, but can only explain less than five percent of variability (adjusted $R^2 = 0.04414$).

DW is negatively correlated with the Code-to-Comment ratio. Therefore, we cannot reject H_6.

5.3 RQ2: Can metrics implemented by RAMA predict functional correctness?

In this section we present the results related to functional correctness (RQ2).

5.3.1 H_7 Projects with poor RAMA metrics have more bugs



(a) Distribution of RAMA metrics and bugfixing commits **(b)** Correlation of RAMA metrics and bugfixing commits

Figure 5.7: Distribution (a) and correlation (b) of RAMA metrics and bugfixing commits

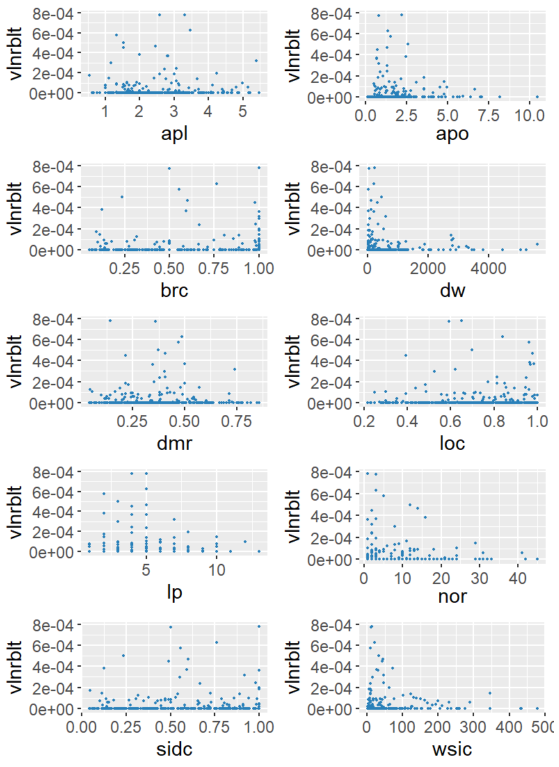
Figure 5.7a shows the distribution of RAMA metrics and bugfixing commits. Most repositories have a bugfixing ratio of less than 40 %. Figure 5.7b shows the correlation of RAMA metrics and bugfixing commits. We removed outliers that distorted the Q-Q plot (top and bottom 1 % of data w.r.t. bugfixing commits). Choosing the significance level of 0.01, no RAMA metric is statistically significantly correlated with the ratio of bugfixing commits. Of all RAMA metrics, APO predicts the source code quality w.r.t. ratio of bugfixing commits best (β = 0.65, p = 0.013). The model has a p-value of 0.000591, but is only able to explain about 5 percent of variability in the data (adjusted R^2 = 0.05814).

Since no RAMA metric is statistically significantly correlated with the percentage of bugfixing commits, we reject H_7

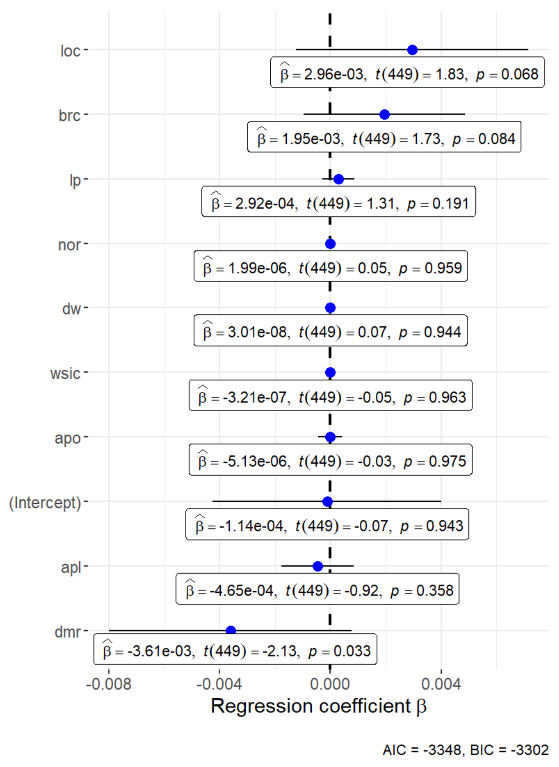
5.4 RQ3: Can metrics implemented by RAMA predict security?

In this section we present the results related to security (RQ3).

5.4.1 H_8 Projects with poor RAMA metrics have more vulnerabilities



(a) Distribution of RAMA metrics and vulnerabilities



(b) Correlation of RAMA metrics and vulnerabilities

Figure 5.8: Distribution (a) and correlation (b) of RAMA metrics and vulnerabilities

Figure 5.8a shows the distribution of RAMA metrics and vulnerabilities. Most repositories have few vulnerabilities and therefore most values are close to zero. Figure 5.8b shows the correlation of RAMA metrics and vulnerabilities. The Q-Q plot shows a right-skewed distribution of residuals due to many vulnerability values having a zero value. We apply the square root transformation to the dependent variable with Shapiro-Wilk's normality test showing a p-value of less than 2.2×10^{-16} after transformation. The model indicates that no RAMA metric correlates with the number of vulnerabilities. The negatively correlated metric DMR ($\beta = -3.61 \times 10^{-03}$) has the most significant correlation with a p-value of 0.033. The model has a p-value of 0.9432 and can only explain less than one percent of variability (adjusted $R^2 = 0.005725$).

Since no RAMA metric is statistically significantly correlated with the number of vulnerabilities, we reject H_8.

5.5 RQ4: How does the prediction evolve over time?

We analyzed 2.762 versions which contained API description files across 105 repositories with the most NCLOC. We analyzed 50 versions for each repository. We applied the criteria defined in Section 4.3.1 to exclude unsuitable repository versions. Figure 5.9 shows the number of repositories with API description files per version. The median version when API description files are introduced to a project is 27.5.

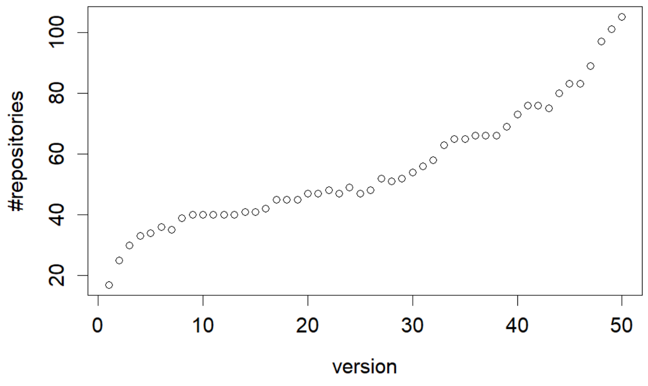


Figure 5.9: Number of repositories per version that contain an API description file

Before investigating the prediction over time, we examine the mean of RAMA metrics (Figure 5.10) and software quality metrics (Figure 5.11) over versions. The trend line is a generalized additive model [Woo21] with a cubic spline smoothing function. The APL tends to decrease first and increase between versions 25 and 30. The values of APO, DW, and DMR steadily decrease over the evolution of a project, while LP and NOR steadily increase. LoC$_{msg}$ quickly reaches a value of approximately 0.7 and remains consistent afterwards. BRC and SIDC show a similar pattern. Their values decrease slightly between version 1 and 30, with an increase in following versions. WSIC's trend shows a consistent value across all versions.

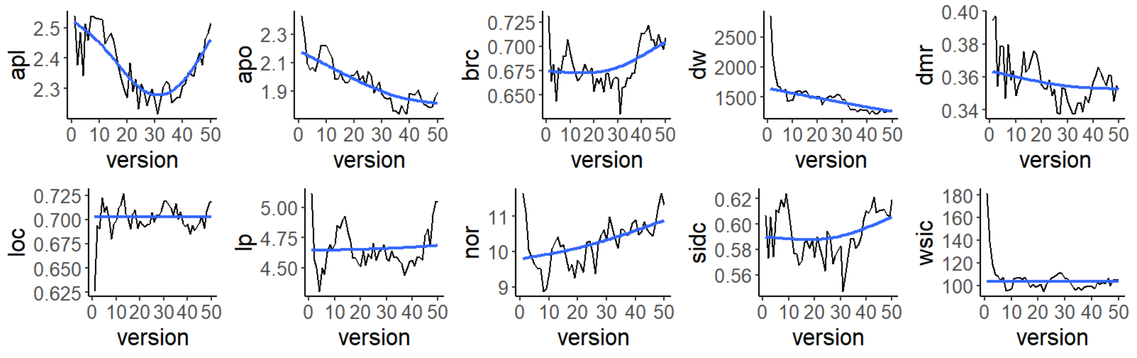


Figure 5.10: Trend of RAMA metrics over versions

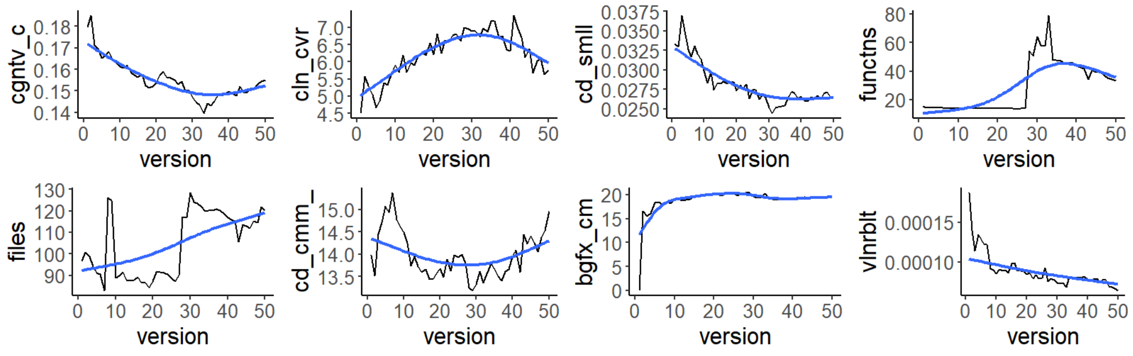


Figure 5.11: Trend of software quality metrics over versions

The values of Cognitive Complexity, Code Smells, and Code-to-Comment ratio decrease between version 1 and 30 and increase in the following versions. The metrics Clone Coverage and average function length show an increasing value up to version 30 and subsequently a slight decrease. While the average file length tends to increase steadily, the number of vulnerabilities per NCLOC tends to decrease. The fraction of bugfixing commits shows a logarithmic trend. Its value increases rapidly in the first five versions and remains almost constant thereafter.

Figure 5.12 shows the trend of Pearson's r coefficient for RAMA metrics and Cognitive Complexity over 50 versions. None of the API metrics is statistically significantly correlated with Cognitive Complexity. Most correlation values show a similar pattern and tend to regress toward zero over the course of project evolution. Exceptions are DW and LoC$_{msg}$. DW's correlation coefficient trend is consistent with an approximate value of -0.12 across all versions. The correlation of LoC$_{msg}$ and Cognitive Complexity increases until version 30 but decreases afterwards.

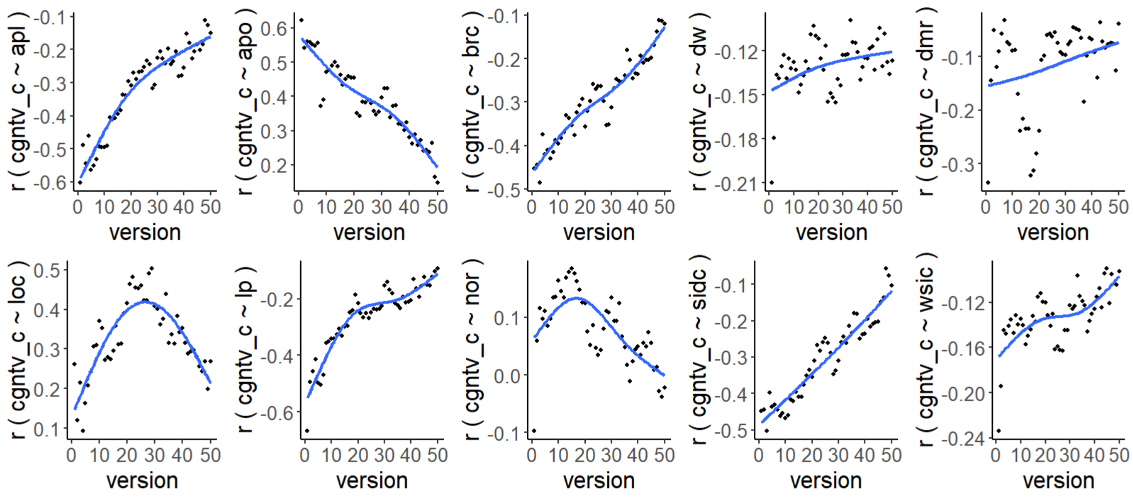


Figure 5.12: Trend of Pearson's r coefficient of RAMA metrics and Cognitive Complexity

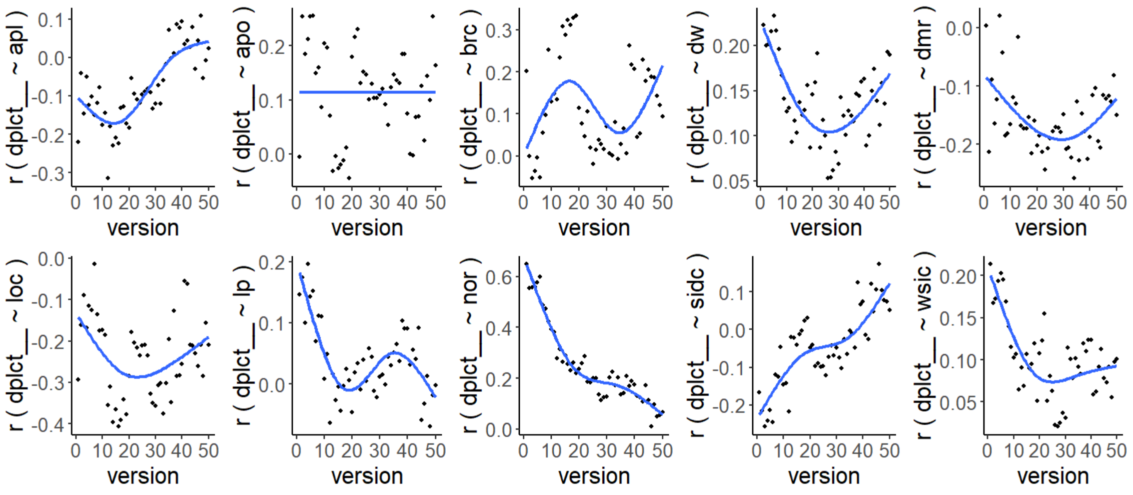


Figure 5.13: Trend of Pearson's r coefficient of RAMA metrics and Clone Coverage

Figure 5.13 shows the correlation of RAMA metrics and Clone Coverage. Similar to Cognitive Complexity, there is no statistically significantly correlated API metric and most correlation values are small or regress toward zero. The trend of BRC shows values around between 0 and 0.3, with increasing values in the last versions. The correlation values of DMR and LoC$_{msg}$ with Clone Coverage tend to increase first but decrease from version 25 to 50.

The trend of Pearson's r coefficient for API metrics and Code Smells is shown in Figure 5.14. DMR correlates statistically significantly with the number of Code Smells (see Section 5.2.3). LP digresses from zero over the evolution of a project. APO regresses toward $r \approx 0.2$. BRC, DW, DMR, and NOR, and SIDC show values near zero for all versions. LoC$_{msg}$ shows a high variance with values between 0 and 0.2 between version 1 and 50.

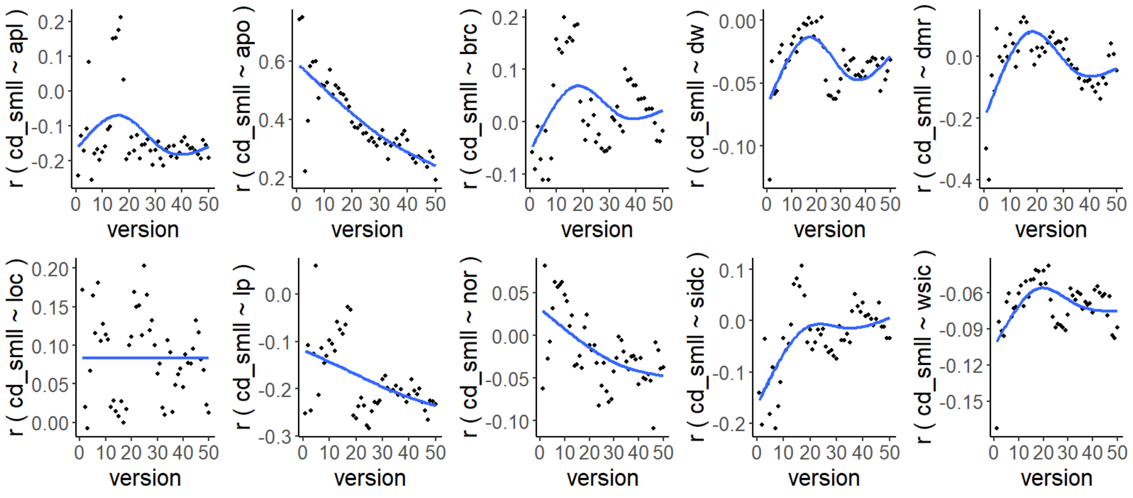


Figure 5.14: Trend of Pearson's r coefficient of RAMA metrics and Code Smells

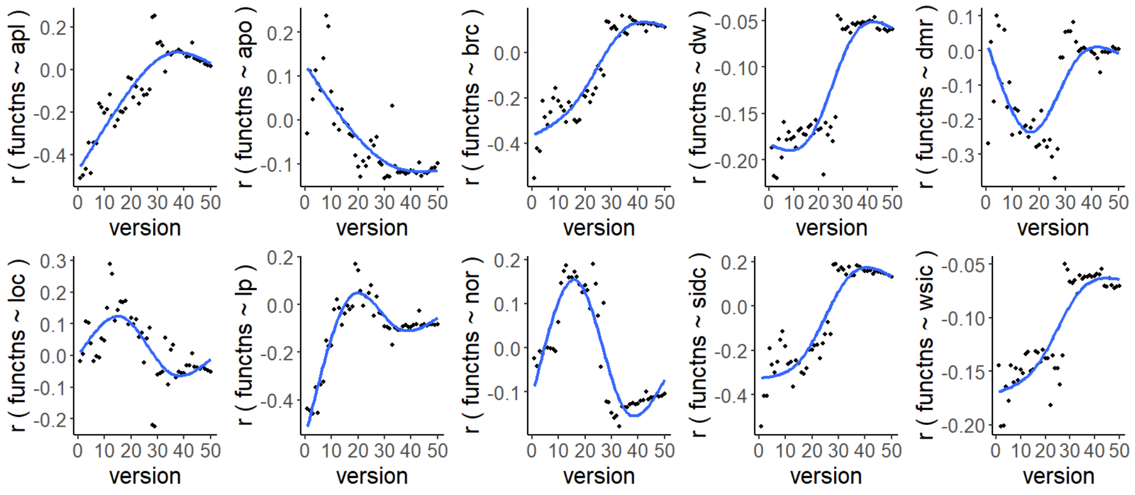


Figure 5.15: Trend of Pearson's r coefficient of RAMA metrics and average function length

Figure 5.15 shows the correlation coefficient of RAMA metrics and average function length. DMR is statistically significantly correlated with the average function length. As the version progresses and more repositories introduce API description files, the correlation coefficient approaches zero for APL, DMR, LoC$_{msg}$, LP. APO and NOR both approach $r = -0.1$, BRC and SIDC $r = 0.1$, and DW and WSIC $r = -0.6$.

Figure 5.16 shows the correlation of RAMA metrics and average file length. The RAMA metric NOR is statistically significantly correlated with the average file length. APL, APO, BRC, DMR, and LoC$_{msg}$ regress toward zero. LP and NOR regress toward $r = -0.1$, DW toward $r = -0.075$, and SIDC toward $r = 0.1$. The correlation of WSIC and average file length decreases and approaches a value of approximately -0.08 over the evolution of a project.

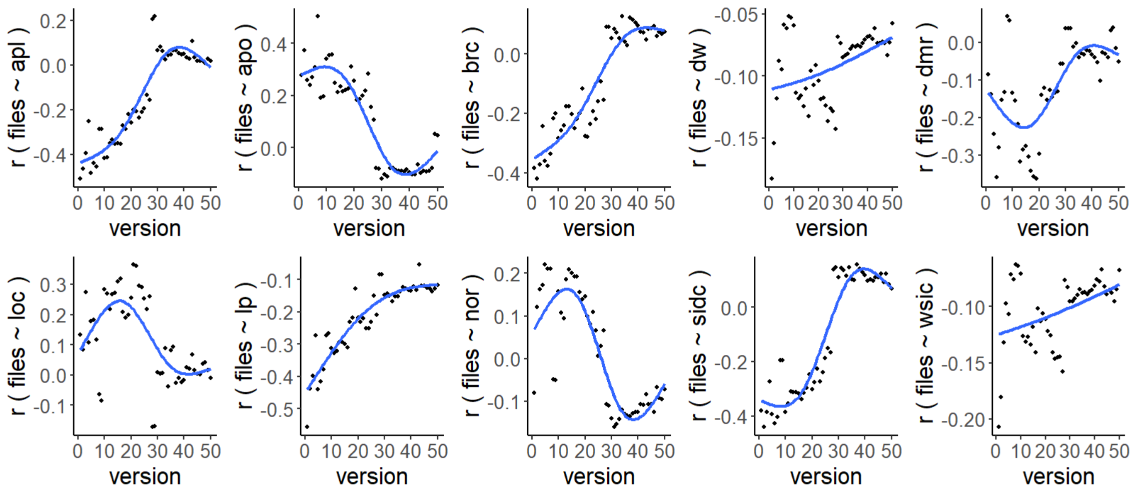


Figure 5.16: Trend of Pearson's r coefficient of RAMA metrics and average file length

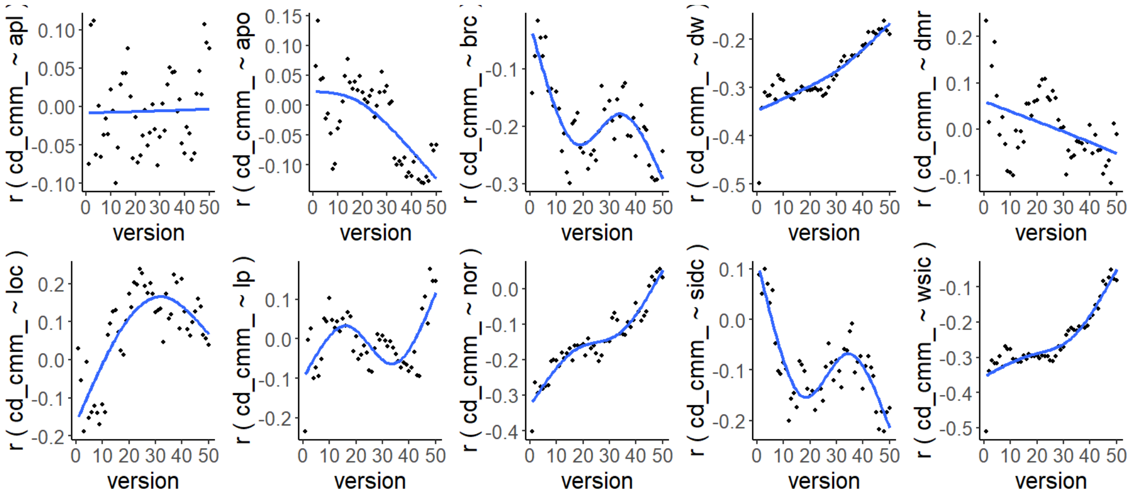


Figure 5.17: Trend of Pearson's r coefficient of RAMA metrics and Code-to-Comment ratio

Figure 5.17 presents Pearson's correlation coefficient r of RAMA metrics and Code-to-Comment ratio. DW is negatively correlated with the Code-to-Comment ratio (see Section 5.2.6). However, the *r* value of DW and Code-to-Comment ratio shows a decreasing correlation over time. DW's correlation decreases toward -0.2 at version 50. APL shows a high variance in correlation that is centered around zero. DMR, NOR, and WSIC show a similar trend of decreasing correlation. APO's correlation coefficient *r* shows values around 0.1 for the first 30 versions. For the following versions, it shows values around -0.1. LP's correlation coefficient shows values around zero with an increase in later versions. The correlations of Code-to-Comment ratio with BRC and SIDC increase. BRC approaches -0.3 and SIDC -0.2 with increasing version and thus a larger sample size per version.

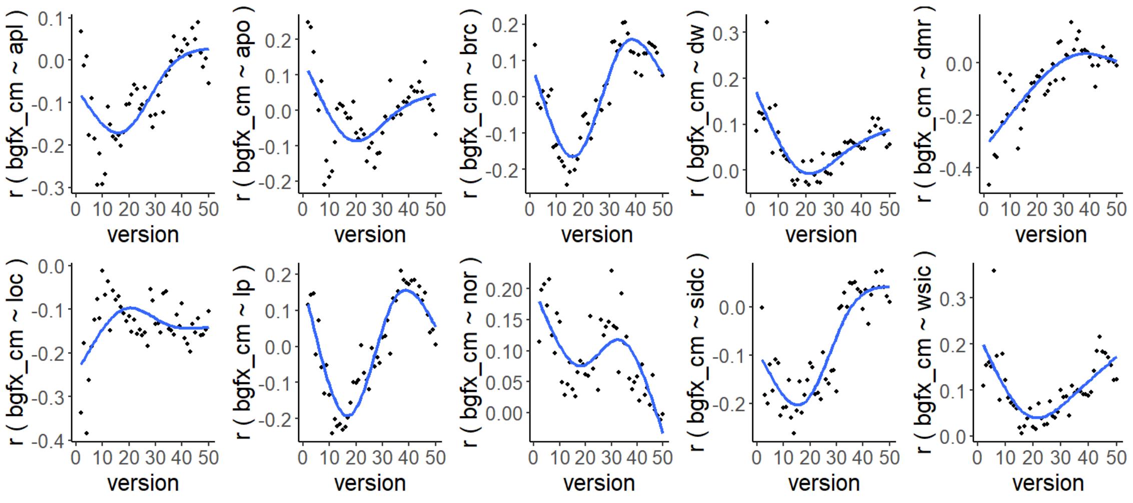


Figure 5.18: Trend of Pearson's r coefficient of RAMA metrics and bugfixing commits

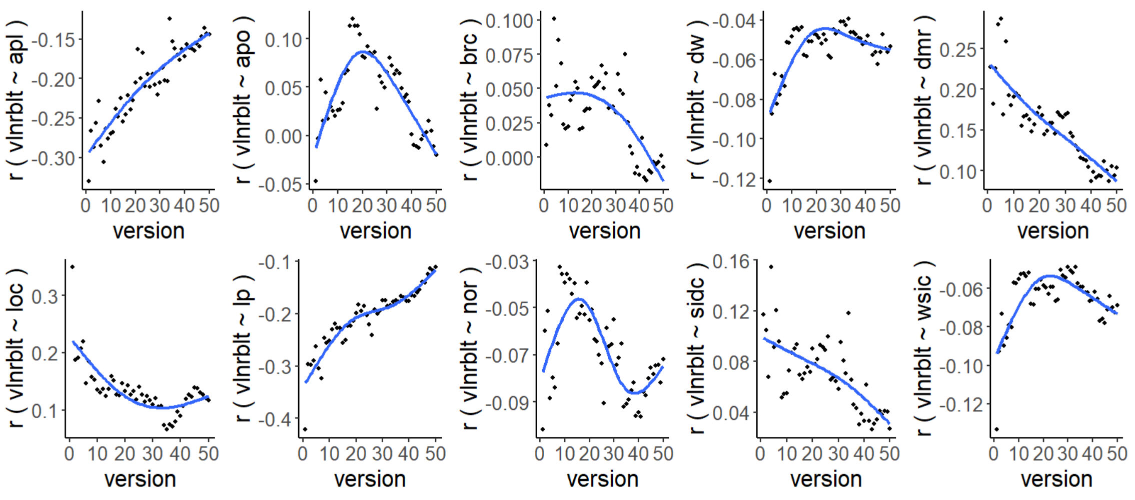


Figure 5.19: Trend of Pearson's r coefficient of RAMA metrics and vulnerabilities

The ratio of bugfixing commits measures the functional correctness of services. The evolution of Pearson's correlation coefficient r is displayed in Figure 5.18. We described in Section 5.3.1 that no RAMA metric is statistically significantly correlated with the ratio of bugfixing commits. The correlation of APL, APO, DMR, NOR, and SIDC regresses toward zero over the evolution. BRC, DW, LP, and WSIC trend to a correlation coefficient of 0.1. LoC$_{msg}$'s correlation coefficient reaches -0.1 with progressing versions.

Figure 5.19 shows Pearson's correlation coefficient r for RAMA metrics and vulnerabilities over 50 versions. No RAMA metric is statistically significantly correlated with the number of vulnerabilities (see Section 5.4.1). Most RAMA metrics show a decreasing correlation over time. APO, BRC approach 0 for later versions, while APL reaches approximately -0.15. DW and WSIC correlation coefficient trends shows values between -0.05 and -0.08. The correlation of DMR and vulnerabilities declines from a value of approximately 0.25 and reaches 0.1 for version 50. LoC$_{msg}$ shows values between 0.20 and 0.1. LP's r value shows a declining correlation with $r > -0.3$ for version 1 and $r \approx -0.1$ for version 50. NOR shows fluctuating values between -0.03 and -0.09. The coefficient of SIDC and vulnerabilities decreases from 0.15 at version 1 to less than 0.04 at version 50.

6 Discussion

This chapter discusses study results, shows implications, and describes threats to validity.

6.1 Results Discussion

RQ1 - RQ3: Our results indicate there are only few correlations between API metrics and our selected software quality metrics. DMR is negatively correlated with the number of Code Smells and positively correlated with the average function length, supporting Hypotheses H_3 and H_4. NOR is positively correlated with the average file length, supporting H_5. Hypothesis H_6 is supported by the negative correlation between DW and Code-to-Comment ratio.

Although the model for Cognitive Complexity did not show a statistically significant p-value for DMR, it is small (p = 0.028). This coincides with the notion of Baski and Misra, that consistently structured messages of operations are easier to remember because they have a similar structure which reduces the complexity of a service [BM11].

Small values of DMR are better [BWZ20]. However, the model in Section 5.2.3 shows a negative correlation between DMR and Code Smells, which contradicts this. A value decrease of 1 for DMR would increase the number of Code Smells per NCLOC by 0.08. Since the values of DMR range between 0 and 1, this effect is probably not noticeable for developers in practice. DMR also has a negative correlation with Clone Coverage (p = 0.022), other correlations with $p < 0.05$ are positive (Cognitive Complexity p = 0.028, Average Function Length p = 0.009) The negative correlations might exist due to the repositories in our sample, or perhaps the notion that smaller values of DMR are better is not totally correct.

One root more in an interface increases the NCLOC per file by 0.05. This effect shown by Figure 5.5b is negligible in practice, since the NOR increases only slightly, as seen in Figure 5.10. When following Hypermedia as the Engine of Application State (HATEOAS) constraints of the REST style, an API should not have more than one root resource [HLV18].

The negative correlation of DW and Code-to-Comment ratio could be explained by the temptation of developers not commenting objects with numerous parameters since it poses much effort. In contrast, objects with few parameters are commented more frequently because it is less effort. However, with an effect size of $\hat{\beta} = -2.86 \times 10^{-04}$ in the linear regression model and $r = -0.0428$ for Pearson's correlation coefficient, the effect of DW on Code-to-Comment ratio is negligible in practice.

Our results regarding WSIC do not coincide with those of the authors of the metric. They observed that the complexity of a service, measured with the SOA Complexity Index (SCI) and Services Complexity Index (SVCI), increases when WSIC increases [HCA09]. We do not have a statistically significant correlation between the complexity metric Cognitive Complexity and WSIC. However, this may be explained by the different operationalization of complexity.

Due to the number of hypotheses, we chose a significance level of 0.01. This strict significance level could have resulted in the exclusion of some correlations that are not apparent due to the sample. Considering p-values < 0.05, DMR is correlated with five of our eight software quality metrics, NOR with three and APO with two. BRC, LP, DW, LoC$_{msg}$, and WSIC have p-values < 0.05 for one of our selected software quality metrics. APL never shows a p-value < 0.05, suggesting it is the least useful RAMA metric.

RQ4: The evolution of the correlation of API metrics and software quality metrics as examined in Section 5.5 show a similar trend for most API metrics. The correlation regresses toward a very small value, sometimes even zero. In addition, all regression models of H_1 - H_8 show a small value for the adjusted R^2. This indicates that as a project evolves, an increasing number of factors besides the API quality influence the quality of the source code.

6.2 Implications

Although our results suggest there is not a large correlation between API quality and software quality, API designers should not ignore best practices and API metrics. Especially in early stages of development, API metrics tend to show a stronger correlation with software quality metrics and should therefore be considered by API designers and developers. When considering the relevance of individual RAMA metrics, DMR seems to be the most useful one, followed by NOR and APO. If the API is designed to be used by third parties, designers should consider that complex and difficult to use APIs lead to more defects in the third party applications [THN20].

6.3 Threats to Validity

Despite taking numerous precautions during study design and data analysis, there are several threats to validity.

6.3.1 Construct validity

The operationalization of software quality is a widely discussed topic. There are several proposed quality models that describe indicators for measuring software quality [HKV07] [WGH+15]. To support the analysis of a large sample of repositories, we limited ourselves to statically measurable metrics. We selected metrics used in previous research that studied maintainability and functional correctness [RVWJ19] [Car11] [Aba21]. However, the used metrics might not cover all aspects of the examined software quality equivalently, such as testability (Section 2.1).

According to Amit and Feitelson, a keyword search for bugfixing commits, which we used in our study, reaches 88% accuracy for their manually classified sample of commits [AF20]. This metric is used in several other studies due to its practicality ([GL17] [Aba21]) and generally accepted by the community [GL17]. A more accurate way to identify bugfixing commits could be the more complex Corrective Commit Probability (CCP) metric proposed by Amit and Feitelson, which is based on mathematical models [AF20].

The adequate detection of vulnerabilities with static tools is difficult. Many tools have a low precision when detecting vulnerabilities in source code and other tools show better precision and might lead to different results than the vulnerability analysis with SonarQube [LLSP21].

6.3.2 Internal validity

We relied on automation for selection and analysis of repositories. While this reduces the problem of human errors and inconsistencies, undetected bugs in our code could affect the results.

Di Lauro et al. suggests that all APIs are not implemented in functioning services [DSP21]. Hence, appropriate exclusion criteria were used as described in Section 4.3.1, and duplicate API description files were detected by filename or identical metric values. The criteria were refined several times using a random selection of repositories. However, incorrectly included/excluded API description files, source code files, or RAMA results could distort the results and obfuscate an existing or show a non-existing correlation. Some API description files could not be converted or analyzed due to syntax errors. If a repository contained multiple API description files, we aggregated the results of all files. Although we detected duplicated API description files and aggregated their metrics accordingly, undetected duplicates might have distorted the aggregated values, especially for DW and WSIC.

We manually examined the commits of more than 20 repositories to validate the accuracy of the keywords used to count the number of bugfixing commits. Some false positives that are not related to source code are fixes in the documentation or references to elements unrelated to source code, such as “Starting project for error handling tutorial video”. There were approximately three false positives in 100 commits.

6.3.3 External validity

As described in Section 4.8, not all programming languages are represented equally in the sample. Since languages differ in aspects such as function length [RVWJ19], the results can be generalized to the most represented languages but might not be applicable for underrepresented languages such as Kotlin or C. For example, to reduce the impact of different languages on the result, we used function length averages instead of counting the number of functions that exceed a certain threshold.

Most analyzed repositories have a small number of publicly exposed interfaces ($median_{WSIC} = 17$) and NCLOC ($median_{NCLOC} = 4,081$). We normalized all concerned software quality metrics by NCLOC to improve the comparability between repositories with few and many NCLOC. However, the results might differ for repositories with bigger interfaces. For our study we prioritized the sample size over stricter inclusion criteria.

Since all analyzed repositories are open-source, results could differ for closed-source code. Closed-source project might have differently experienced developers as well as quality assurance and development processes.

6.3.4 Conclusion validity

We use a linear model for the statistical analysis, assuming there is a linear relation between independent and dependent variables. Linear models are not suitable if there is a non-linear relation, such as logarithmic or exponential relations. However, RAMA metrics of our sample data do not display a clear, non-linear relation to a software quality metric. Furthermore, we considered assumptions of multiple linear regression such as heteroskedasticity or autocorrelation and examined our sample data accordingly. Repositories with any missing value for API or software metric were not included in the statistical analysis. As described in Section 4.6.2, we removed SIDC from the model due to multicollinearity with BRC. However, the significance of variables in the regression model could be distorted by multicollinearity between other RAMA metrics, although their VIF showed values < 10. If the assumption of multivariate normality was violated, we transformed the dependent variable using the square root transformation. The result did not correct non-normality of residuals for the models of all software quality metrics. This might affect the significance values of the models.

6.3.5 Replicability

Most steps of the repository selection, static and statistical analysis process are automated to allow for straightforward replication of the study. We performed manual repository selection in addition to automatic repository selection. When replicating the study, the results might differ due to the manual selection of repositories. Moreover, developers will have pushed newer commits to the git repository, possibly changing API and source code.

7 Conclusion

To evaluate whether API metrics implemented by RAMA can effectively evaluate the quality of RESTful APIs, we conducted a large scale MSR study and compared API metrics and software quality metrics. We analyzed more than 440 repositories which contain machine-readable API documentation and source-code implementing the APIs. The API metrics were computed by RAMA and software quality metrics were provided by SonarQube and git. The metrics were examined with multiple linear regression to determine how they are correlated. Our results suggest that DMR is statistically significantly correlated with the number of Code Smells and the average function length, NOR with the average file length, and DW with Code-to-Comment ratio. However, the effect sizes are small and the regression models cannot explain much variability in the data. Furthermore, we investigated how the correlation of API metrics and software quality metrics changes over the evolution of a project. The results show that effect sizes of the correlation of most API metrics and software quality metrics decrease with newer versions. This could be explained by the fact that in the course of development other factors that influence software quality weigh more heavily than the quality of the API design.

We conclude that the proposed service interface metrics measure the quality of RESTful APIs to a certain degree. Our approach showed that the API metrics cannot predict the quality of services implementing the interfaces with certainty. However, API quality correlates with software quality to some extent, especially in terms of maintainability. Hence, API designers and developers should consider the proposed metrics in early stages of development when designing new APIs.

Future work should focus on the applicability of RESTful API metrics for practitioners. For example, it should study how developers perceive and understand APIs with good and bad API metric values. Second, the relation between RAMA metrics and best practices for APIs should be examined to see whether APIs with good RAMA metrics tend to follow best practices, whereas APIs with bad RAMA metrics ignore best practices and show antipatterns. An approach to measure antipatterns in RESTful APIs was presented by Palma et al. [PGM+15]. Third, it should investigate which factors improve the prediction of software quality. Our models could only explain little variability in the data, i.e. there are other predictors besides API metrics that explain software quality metrics. Fourth, it should compare results of open-source and closed-source code. We only considered open-source repositories. Fifth, it should examine whether the results change when a different operationalization for software quality is used.

Bibliography

[Aba21] D. Abajirov. “Does Functional Programming Improve Software Quality? An Empirical Analysis of Open Source Projects on GitHub”. Bachelor Thesis. 2021 (cit. on pp. 32, 33, 35, 56, 57).

[AF20] I. Amit, D. G. Feitelson. *The Corrective Commit Probability Code Quality Metric*. 2020. arXiv: 2007.10912 [cs.SE] (cit. on p. 57).

[AG96] M. Aickin, H. Gensler. “Adjusting for multiple testing when reporting research results: the Bonferroni vs Holm methods.” In: *American Journal of Public Health* 86.5 (May 1996), pp. 726–728. DOI: 10.2105/ajph.86.5.726. URL: https://doi.org/10.2105/ajph.86.5.726 (cit. on p. 36).

[Alr19] H. Alrubaye. “How Does API Migration Impact Software Quality and Comprehension? An Empirical Study”. In: *CoRR* abs/1907.07724 (2019). arXiv: 1907.07724. URL: http://arxiv.org/abs/1907.07724 (cit. on p. 26).

[AZM+15] D. Athanasopoulos, A. V. Zarras, G. Miskos, V. Issarny, P. Vassiliadis. “Cohesion-Driven Decomposition of Service Interfaces without Access to Source Code”. In: *IEEE Transactions on Services Computing* 8.4 (July 2015), pp. 550–562. DOI: 10.1109/tsc.2014.2310195. URL: https://doi.org/10.1109/tsc.2014.2310195 (cit. on p. 24).

[BFWZ19] J. Bogner, J. Fritzsch, S. Wagner, A. Zimmermann. “Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality”. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2019, pp. 187–195. DOI: 10.1109/ICSA-C.2019.00041 (cit. on p. 15).

[Bie15] M. Biehl. *What are API Description Languages?* June 2015. URL: https://api-university.com/blog/what-are-api-description-languages/ (cit. on p. 20).

[BM09] D. Basci, S. Misra. “Data Complexity Metrics for XML Web Services”. In: *Advances in Electrical and Computer Engineering* 9.2 (2009), pp. 9–15. DOI: 10.4316/aece.2009.02002. URL: https://doi.org/10.4316/aece.2009.02002 (cit. on pp. 23, 24).

[BM11] D. Baski, S. Misra. “Metrics suite for maintainability of eXtensible Markup Language web services”. In: *IET Software* 5.3 (2011), p. 320. DOI: 10.1049/iet-sen.2010.0089. URL: https://doi.org/10.1049/iet-sen.2010.0089 (cit. on pp. 24, 55).

[Bog20] J. Bogner. “On the Evolvability Assurance of Microservices: Metrics, Scenarios, and Patterns”. en. In: (2020). DOI: 10.13140/RG.2.2.10778.67523. URL: http://rgdoi.net/10.13140/RG.2.2.10778.67523 (cit. on pp. 15, 23, 24, 27).

[BP79] T. S. Breusch, A. R. Pagan. “A simple test for heteroscedasticity and random coefficient variation”. In: *Econometrica: Journal of the econometric society* (1979), pp. 1287–1294 (cit. on p. 35).

[BWW20] M. M. Barón, M. Wyrich, S. Wagner. "An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability". In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, Oct. 2020. DOI: 10.1145/3382494.3410636. URL: https://doi.org/10.1145/3382494.3410636 (cit. on p. 32).

[BWZ20] J. Bogner, S. Wagner, A. Zimmermann. "Collecting Service-Based Maintainability Metrics from RESTful API Descriptions: Static Analysis and Threshold Derivation". In: *Communications in Computer and Information Science*. Springer International Publishing, 2020, pp. 215–227. DOI: 10.1007/978-3-030-59155-7_16. URL: https://doi.org/10.1007/978-3-030-59155-7_16 (cit. on pp. 23, 40, 55).

[Cam21] G. A. Campbell. *Cognitive Complexity - A new way of measuring understandability*. 2021. URL: https://www.sonarsource.com/resources/white-papers/cognitive-complexity/ (cit. on p. 21).

[Car11] R. Cardell-Oliver. "How can software metrics help novice programmers?" In: *Proceedings of the Thirteenth Australasian Computing Education Conference-Volume 114*. 2011, pp. 55–62 (cit. on p. 56).

[Cho21] G. Choueiry. *Square Root Transformation: A Beginner's Guide*. 2021. URL: https://quantifyinghealth.com/square-root-transformation/ (cit. on p. 35).

[Coh88] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences (2nd Edition)*. English. Routledge, 1988, p. 400. ISBN: 978-0805802832 (cit. on p. 43).

[DSP21] F. Di Lauro, S. Serbout, C. Pautasso. "Towards Large-Scale Empirical Assessment of Web APIs Evolution". In: *Web Engineering*. Ed. by M. Brambilla, R. Chbeir, F. Frasincar, I. Manolescu. Cham: Springer International Publishing, 2021, pp. 124–138. ISBN: 978-3-030-74296-6 (cit. on pp. 26, 57).

[EB18] A. M. Eilertsen, A. H. Bagge. "Exploring API". In: *Proceedings of the 2nd International Workshop on API Usage and Evolution*. ACM, June 2018. DOI: 10.1145/3194793.3194799. URL: https://doi.org/10.1145/3194793.3194799 (cit. on p. 26).

[ESE19] ESE Group, University of Stuttgart. *RAMA CLI*. 2019. URL: https://github.com/restful-ma/rama-cli/ (cit. on pp. 24, 25).

[Fie08] R. T. Fielding. *REST APIs must be hypertext-driven*. 2008. URL: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (cit. on p. 19).

[Fre07] Free Software Foundation. *GNU Lesser General Public License*. 2007. URL: https://www.gnu.org/licenses/lgpl-3.0.en.html (cit. on p. 21).

[FS15] W. Fenske, S. Schulze. "Code smells revisited: A variability perspective". In: *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*. 2015, pp. 3–10 (cit. on p. 32).

[GGS+15] P. Giessler, M. Gebhart, D. Sarancin, R. Steinegger, S. Abeck. "Best practices for the design of restful web services". In: *International Conferences of Software Advances (ICSEA)*. 2015, pp. 392–397 (cit. on pp. 18–20).

[Git21] GitHub, Inc. *GitHub REST API*. 2021. URL: https://docs.github.com/en/rest (cit. on pp. 29, 31, 36, 37).

[GL17] Y. Gil, G. Lalouche. “On the correlation between size and metric validity”. In: 22.5 (June 2017), pp. 2585–2611. DOI: 10.1007/s10664-017-9513-5. URL: https://doi.org/10.1007/s10664-017-9513-5 (cit. on p. 57).

[Gos20] T. Goswami. *Difference between WSDL and WADL*. Sept. 2020. URL: https://www.programsbuzz.com/article/difference-between-wsdl-and-wadl (cit. on p. 20).

[Gou13] G. Gousios. “The GHTorent Dataset and Tool Suite”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 233–236. ISBN: 9781467329361 (cit. on p. 29).

[HCA09] M. Hirzalla, J. Cleland-Huang, A. Arsanjani. “A Metrics Suite for Evaluating Flexibility and Complexity in Service Oriented Architectures”. In: *Service-Oriented Computing – ICSOC 2008 Workshops*. Springer Berlin Heidelberg, 2009, pp. 41–52. DOI: 10.1007/978-3-642-01247-1_5. URL: https://doi.org/10.1007/978-3-642-01247-1_5 (cit. on pp. 25, 56).

[HKV07] I. Heitlager, T. Kuipers, J. Visser. “A Practical Model for Measuring Maintainability”. In: *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*. IEEE, Sept. 2007. DOI: 10.1109/quatic.2007.8. URL: https://doi.org/10.1109/quatic.2007.8 (cit. on p. 56).

[HLSV17] F. Haupt, F. Leymann, A. Scherer, K. Vukojevic-Haupt. “A Framework for the Structural Analysis of REST APIs”. In: *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, Apr. 2017, pp. 55–58. DOI: 10.1109/icsa.2017.40. URL: https://doi.org/10.1109/icsa.2017.40 (cit. on pp. 23, 24).

[HLV18] F. Haupt, F. Leymann, K. Vukojevic-Haupt. “API governance support through the structural analysis of REST APIs”. In: *Computer Science - Research and Development* 33 (Aug. 2018). DOI: 10.1007/s00450-017-0384-1 (cit. on p. 55).

[Hol79] S. Holm. “A Simple Sequentially Rejective Multiple Test Procedure”. In: *Scandinavian Journal of Statistics* 6.2 (1979), pp. 65–70. ISSN: 03036898, 14679469. URL: http://www.jstor.org/stable/4615733 (cit. on p. 36).

[ISO01] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001 (cit. on p. 26).

[ISO11] ISO/IEC. *ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. 2011 (cit. on pp. 17, 26).

[ISO14] ISO/IEC. *ISO/IEC 25000:2014, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE)*. 2014. URL: https://iso25000.com/index.php/en/iso-25000-standards (cit. on p. 26).

[KCK11] M. Kim, D. Cai, S. Kim. “An empirical investigation into the role of API-level refactorings during software evolution”. In: *Proceeding of the 33rd international conference on Software engineering - ICSE ’11*. ACM Press, 2011. DOI: 10.1145/1985793.1985815. URL: https://doi.org/10.1145/1985793.1985815 (cit. on p. 26).

[KEE12] Y. A. Khan, M. O. Elish, M. El-Attar. "A Systematic Review on the Impact of CK Metrics on the Functional Correctness of Object-Oriented Classes". In: *Computational Science and Its Applications – ICCSA 2012*. Springer Berlin Heidelberg, 2012, pp. 258–273. DOI: 10.1007/978-3-642-31128-4_19. URL: https://doi.org/10.1007/978-3-642-31128-4_19 (cit. on p. 27).

[KGB+14] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian. "The Promises and Perils of Mining GitHub". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 92–101. ISBN: 9781450328630. DOI: 10.1145/2597073.2597074. URL: https://doi.org/10.1145/2597073.2597074 (cit. on p. 36).

[KK18] I. Koren, R. Klamma. "Enabling visual community learning analytics with Internet of Things devices". In: *Computers in Human Behavior* 89 (Dec. 2018), pp. 385–394. DOI: 10.1016/j.chb.2018.07.036. URL: https://doi.org/10.1016/j.chb.2018.07.036 (cit. on p. 21).

[Lin21] Linux Foundation. *The OpenAPI Specification*. 2021. URL: https://github.com/OAI/OpenAPI-Specification (cit. on p. 25).

[LLSP21] V. Lenarduzzi, S. Lujan, N. Saarimaki, F. Palomba. "A Critical Comparison on Six Static Analysis Tools: Detection, Agreement, and Precision". In: *arXiv preprint arXiv:2101.08832* (2021) (cit. on p. 57).

[LS04] A. Langhorst, M. Steinle. "Pipes and Filters Architectural Pattern". In: *Technische Berichte* (2004), p. 3 (cit. on p. 25).

[MICV17] N. Medeiros, N. Ivaki, P. Costa, M. Vieira. "Software Metrics as Indicators of Security Vulnerabilities". In: *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Oct. 2017. DOI: 10.1109/issre.2017.11. URL: https://doi.org/10.1109/issre.2017.11 (cit. on pp. 27, 28).

[MPD10] M. Maleshkova, C. Pedrinaci, J. Domingue. "Investigating Web APIs on the World Wide Web". In: *2010 Eighth IEEE European Conference on Web Services*. IEEE, Dec. 2010. DOI: 10.1109/ecows.2010.9. URL: https://doi.org/10.1109/ecows.2010.9 (cit. on p. 18).

[Mul12] B. Mulloy. *Web API Design: Crafting Interfaces that Developers Love*. 2012 (cit. on p. 18).

[NMM+99] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/RFC2616. URL: https://rfc-editor.org/rfc/rfc2616.txt (cit. on p. 19).

[NNR19] P. Nistala, K. V. Nori, R. Reddy. "Software Quality Models: A Systematic Mapping Study". In: *2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)*. IEEE, May 2019. DOI: 10.1109/icssp.2019.00025. URL: https://doi.org/10.1109/icssp.2019.00025 (cit. on p. 26).

[OW14] J.-P. Ostberg, S. Wagner. "On Automatically Collectable Metrics for Software Maintainability Evaluation". In: *2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*. IEEE, Oct. 2014. DOI: 10.1109/iwsm.mensura.2014.19. URL: https://doi.org/10.1109/iwsm.mensura.2014.19 (cit. on pp. 22, 32, 33).

[OWA17] OWASP Foundation. *OWASP Top Ten 2017*. 2017. URL: https://owasp.org/www-project-top-ten/2017/ (cit. on p. 22).

[PGM+15] F. Palma, J. Gonzalez-Huerta, N. Moha, Y.-G. Guéhéneuc, G. Tremblay. “Are RESTful APIs Well-Designed? Detection of their Linguistic (Anti)Patterns”. In: *Service-Oriented Computing*. Springer Berlin Heidelberg, 2015, pp. 171–187. DOI: 10.1007/978-3-662-48616-0_11. URL: https://doi.org/10.1007/978-3-662-48616-0_11 (cit. on p. 59).

[PNIR20] A. A. Prayogi, M. Niswar, Indrabayu, M. Rijal. “Design and Implementation of REST API for Academic Information System”. In: *IOP Conference Series: Materials Science and Engineering* 875 (July 2020), p. 012047. DOI: 10.1088/1757-899x/875/1/012047. URL: https://doi.org/10.1088/1757-899x/875/1/012047 (cit. on p. 18).

[PRF07] M. Perepletchikov, C. Ryan, K. Frampton. “Cohesion Metrics for Predicting Maintainability of Service-Oriented Software”. In: *Seventh International Conference on Quality Software (QSIC 2007)*. IEEE, 2007. DOI: 10.1109/qsic.2007.4385516. URL: https://doi.org/10.1109/qsic.2007.4385516 (cit. on p. 25).

[RAM16] RAML Workgroup. *RAML Version 1.0: RESTful API Modeling Language*. May 2016. URL: https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/ (cit. on p. 25).

[Rob09] M. P. Robillard. “What Makes APIs Hard to Learn? Answers from Developers”. In: 26.6 (Nov. 2009), pp. 27–34. DOI: 10.1109/ms.2009.193. URL: https://doi.org/10.1109/ms.2009.193 (cit. on p. 15).

[RVWJ19] T. Roehm, D. Veihelmann, S. Wagner, E. Juergens. “Evaluating Maintainability Prejudices with a Large-Scale Study of Open-Source Projects”. In: *Software Quality: The Complexity and Challenges of Software Engineering and Software Quality in the Cloud*. Ed. by D. Winkler, S. Biffl, J. Bergsmann. Cham: Springer International Publishing, 2019, pp. 151–171. ISBN: 978-3-030-05767-1 (cit. on pp. 31, 32, 56, 57).

[SAN11] SANS^{TM} Foundation. *CWE/SANS TOP 25 Most Dangerous Software Errors*. 2011. URL: https://www.sans.org/top25-software-errors/ (cit. on p. 22).

[Sch16] A. Scherer. “Description Languages for REST APIs - State of the Art, Comparison, and Transformation”. Master Thesis. 2016 (cit. on p. 21).

[Sch21] S. Schneider. *Can Proposed Service Interface Metrics Effectively Evaluate the Quality of RESTful APIs? Dataset and Filter Criteria*. Zenodo, Oct. 2021. DOI: 10.5281/zenodo.5559462. URL: https://doi.org/10.5281/zenodo.5559462 (cit. on p. 32).

[SCL16] G. Schermann, J. Cito, P. Leitner. “All the Services Large and Micro: Revisiting Industrial Practice in Services Computing”. In: *Service-Oriented Computing – ICSOC 2015 Workshops*. Ed. by A. Norta, W. Gaaloul, G. R. Gangadharan, H. K. Dam. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 36–47. ISBN: 978-3-662-50539-7 (cit. on p. 15).

[SD15] D. Steidl, F. Deissenboeck. “How do Java methods grow?” In: *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, Sept. 2015. DOI: 10.1109/scam.2015.7335411. URL: https://doi.org/10.1109/scam.2015.7335411 (cit. on pp. 31, 32).

[Sma21] SmartBear Software. *Swagger Specification*. 2021. URL: https://swagger.io/docs/specification (cit. on p. 23).

[Son21a] SonarSource S.A. *Languages - Overview*. 2021. URL: https://docs.sonarqube.org/latest/analysis/languages/overview/ (cit. on p. 30).

[Son21b] SonarSource S.A. *Metric Definitions*. 2021. URL: https://docs.sonarqube.org/latest/user-guide/metric-definitions/ (cit. on pp. 28, 32–35).

[Son21c] SonarSource S.A. *Rules*. 2021. URL: https://docs.sonarqube.org/latest/user-guide/rules/ (cit. on p. 22).

[Son21d] SonarSource S.A. *Security-related Rules*. 2021. URL: https://docs.sonarqube.org/latest/user-guide/security-rules/ (cit. on p. 22).

[Son21e] SonarSource S.A. *SonarQube*. 2021. URL: https://www.sonarqube.org/ (cit. on p. 21).

[Sun09] Sun Microsystems. *Web Application Description Language*. 2009. URL: https://www.w3.org/Submission/wadl/ (cit. on pp. 20, 25).

[Tam17] T. Tam. *MuleSoft Joins the OpenAPI Initiative: The End of the API Spec Wars*. 2017. URL: https://swagger.io/blog/news/mulesoft-joins-the-openapi-initiative/ (cit. on p. 21).

[The21] The MITRE Corporation. *List of Weaknesses*. 2021. URL: https://cwe.mitre.org/ (cit. on pp. 22, 34).

[THN20] H. Tahmooresi, A. Heydarnoori, R. Nadri. "Studying the Relationship Between the Usage of APIs Discussed in the Crowd and Post-Release Defects". In: *Journal of Systems and Software* 170 (Dec. 2020), p. 110724. DOI: 10.1016/j.jss.2020.110724. URL: https://doi.org/10.1016/j.jss.2020.110724 (cit. on pp. 26, 56).

[TJL17] D. Taibi, A. Janes, V. Lenarduzzi. "How developers perceive smells in source code: A replicated study". In: *Information and Software Technology* 92 (Dec. 2017), pp. 223–235. DOI: 10.1016/j.infsof.2017.08.008. URL: https://doi.org/10.1016/j.infsof.2017.08.008 (cit. on p. 33).

[WGH+15] S. Wagner, A. Goeb, L. Heinemann, M. Kläs, C. Lampasona, K. Lochmann, A. Mayr, R. Plösch, A. Seidl, J. Streit, A. Trendowicz. "Operationalised product quality models and assessment: The Quamoco approach". In: *Information and Software Technology* 62 (June 2015), pp. 101–123. DOI: 10.1016/j.infsof.2015.02.009. URL: https://doi.org/10.1016/j.infsof.2015.02.009 (cit. on pp. 26, 56).

[Woo21] S. Wood. *gam: Generalized additive models with integrated smoothness estimation*. 2021. URL: https://www.rdocumentation.org/packages/mgcv/versions/1.8-36/topics/gam (cit. on p. 49).

[WPR10] J. Webber, S. Parastatidis, I. Robinson. *REST in Practice: Hypermedia and Systems Architecture*. Beijing: O'Reilly, 2010. ISBN: 978-0-596-80582-1. URL: https://www.safaribooksonline.com/library/view/rest-in-practice/9781449383312/ (cit. on p. 18).

[Yea21] D. J. Yearsley. *Quantile-Quantile Plots*. 2021. URL: https://www.ucd.ie/ecomodel/Resources/QQplots_WebVersion.html (cit. on p. 35).

All links were last followed on October 10, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature