WEBAPIK: A Body of Structured Knowledge on Designing Web APIs

Mahsa H. Sadi
Microsoft[1]
mahsasadi@microsoft.com

Eric Yu
Department of Computer Science, University of Toronto
eric@cs.toronto.edu

Abstract

With the rise in initiatives such as software ecosystems and Internet of Things (IoT), developing robust web Application Programming Interfaces (web APIs) has become an increasingly important practice. One main concern in developing web APIs is that they expose back-end systems and data towards clients. This exposure threatens critical non-functional requirements, such as the security of back-end systems, the performance of provided services, and the privacy of communications with clients. Although dealing with non-functional requirements during software design has been long studied, there is still little guide on addressing these requirements in web APIs. In this paper, we present WEBAPIK, a body of structured knowledge on addressing non-functional requirements in the design of web APIs. WEBAPIK is comprised of 27 distinct non-functional requirements, 37 distinct design techniques to address some of the identified requirements, and the trade-offs of 22 design techniques, presented in two forms of natural language and knowledge graphs. The design knowledge compiled in WEBAPIK is systematically extracted and aggregated from 80 heterogenous online literature resources, including 7 books, 15 weblogs and tutorial, 5 vendor white papers, 6 design standards, and 47 research papers. These resources are systematically retrieved from two search engines of Google and Google Scholar and five research databases of Web of Science, IEEE Xplore, ACM Digital Library, SpringerLink, and ScienceDirect in two periods of March to August 2018 and August 2022. WEBAPIK gathers and structures expert and scholarly discussions to provide insight about addressing non-functional requirements in the design of web APIs. The structure brought to the design knowledge makes it amenable towards extension and creates the potential for employing it in the database of knowledge-based systems that aids software developers in design decision making.

Keywords

Web APIs, Non-Functional Requirements, Quality Attributes, Design Patterns, Trade-Offs, Software Design, Software Architecture, Knowledge Reuse, Systematic Review

---

[1] The bulk of the reported research has been conducted when the first author was affiliated with the University of Toronto.

1 Introduction

With the rise in initiatives such as software ecosystems [1],[2] and Internet of Things (IoT) [3] in the past decade, developing web Application Programming Interfaces (web APIs) has become an increasingly common practice. Web APIs are programmatic endpoints that make software services accessible over the internet. However, they also expose backend systems and data towards various clients. This exposure raises serious concerns about critical non-functional requirements, such as security of the backend systems, confidentiality and privacy of the exchanged data, and performance of the provided services [4], [5], [6], [7]. These concerns are aggravated considering that fulfilling non-functional requirements are often a matter of trade-offs. Often mechanisms designed to address or improve one non-functional requirement in APIs (such as security) may deteriorate others (such as usability and performance) [8], [9]. When selecting these design mechanisms, developers often need to make difficult trade-offs between various non-functional requirements, of some of which they may be unaware.

Currently, knowledge about addressing non-functional requirements in the design of web APIs is scattered among various heterogeneous online literature resources, including books, vendor white papers, weblogs, and tutorials written by practitioners involved in developing web APIs (e.g., [10], [11], [12], [13], [14]), or disparate bodies of design standards and scholarly articles (e.g., [15], [16], [17], [18], [19], [20], [21]). Requirements and design techniques mentioned in these resources are referred to with different terms and are of different levels of scientific validity. Developers in need of this information, would need to spend considerable amount of time to search and find the related knowledge sources, sift through the retrieved resources based on some criteria, curate information from multiple sources and finally draw conclusions based on the research that they have performed. This problem is exacerbated considering that every now and then, new literature resources are made available by practitioners and scientific community either introducing new design techniques or patterns to address a specific non-functional requirement or discussing and analysing the effect of a design technique on some non-functional requirements.

Nevertheless, knowledge about addressing non-functional requirements in the design of web APIs is highly reusable. Unlike functional requirements whose realization is tightly intertwined with specific structural and procedural properties of the domain under design, mechanisms adopted to realize non-functional requirements in web APIs are rather domain-independent and pattern-like. These mechanisms are often used with minor or without any modifications in various development domains. For example, API-Key [11] is a mechanism to identify and authorize software clients to access a web API. This mechanism can be used to allow access to a flight API that provides the timetable of a given flight; a weather API that provides the weather forecast of a given city; a text summarization API that provides a short summery of a given document; or a logistic regression API that learns to classify a given data set. However, despite the high potential for frequent reuse, very little research has been devoted to assisting software developers in obtaining and using the available design knowledge.

To address the above gap, in this paper, we present WEBAPIK, a body of structured knowledge about addressing non-functional requirements in the design of web APIs. WEBAPIK is comprised of three components: (1) 27 distinct non-functional requirements that should be considered in the design of web APIs, (b) 37 distinct design techniques to address some of the identified non-functional requirements, and (c) the trade-offs 22 design techniques against some non-functional requirements.

The design knowledge compiled in WEBAPIK is systematically extracted and aggregated from 80 heterogenous online literature resources, including 7 books, 15 weblogs and tutorial, 5 vendor white papers, 6 design standards, and 47 research papers.

To collect and organize the design knowledge, we have performed three steps: (1) We have conducted a systematic [22] and evidence-based review [23] of the literature to collect the related knowledge sources. (2) We have extracted the relevant pieces of design knowledge from the collected resources using qualitative text analysis techniques [25], [26], [27], and aggregated and structured them into three categories of (a) non-functional requirements, (b) design techniques to address the identified non-functional requirements, and (c) trade-offs of the design techniques against the identified non-functional requirements, using specific templates. (3) We have summarized and visualized the structured design knowledge using knowledge graphs [28].

The rest of this paper is organized as follows: In Section 2, we explain the methodology for collecting, extracting, and organizing the design knowledge. In Section 3, we present the identified non-functional requirements. In Section 4, we present extracted the design techniques. In Section 5, we present the trade-offs that should be made in selection some of the identified design techniques. In Section 6, we discuss our findings from the performed study and in Section 7, we discuss threats to the validity and reliability of the presented design knowledge. In Section 8, we review the work related to this study. Finally, in Section 9, we summarize and conclude the presented research work.

2 Methodology for Collecting and Organizing the Design Knowledge

The steps performed to collect, extract, and visualize the design knowledge compiled in WEBAPIK are described in the following.

2.1 Collecting Knowledge Sources

The related knowledge resources are collected and selected via a systematic and evidence-based review procedure as recommended in [22], [23]. The details of the performed steps are explained in the following.

2.1.1 Research Questions

The research questions driving the collection of the resources were as follows:

- *RQ 1.* What non-functional requirements need to be considered in designing APIs?
- *RQ 2.* What mechanisms and techniques are suggested and used to address the identified non-functional requirements in the design of APIs?
- *RQ 3.* What are the trade-offs in selecting the identified design techniques?

2.1.2 The Type and Topic of Available Resources

The knowledge addressing the above research questions appears in two main types:

- *Expert discussions*: These discussions appear in the form of books, informal online vendor white papers, weblogs, text and video tutorials, or design standards, are made by practitioners and include their advice, experience and opinion about design techniques and practices.
- *Scholarly discussions*: These discussions appear in the form of journal, conference, and workshop papers, and are written by scholars and researchers. This kind of resources mostly suggest or evaluate design techniques using an analytical or experimental approach.

The resources that address the above research questions include the following topics: (a) "API Design", (b) "API Development", (c) "API management", (d) "API architecture".

2.1.3 Search Process

We have only searched online resource. To gather the resources, we have used two web search engines and five research databases: We have used Google Scholar to find an initial set of books and scholarly articles and have used Google to find informal expert discussions (i.e., weblogs, tutorials, and white papers). To extend the search results for scholarly articles, we have also searched five popular Computer Science and Engineering databases of Web of Science, IEEE Xplore, ACM Digital Library, SpringerLink, and ScienceDirect.

The search process has been performed in several rounds and has been stopped once reached to saturation, i.e., no new resource was found that could be selected. In the first round, a set of resources were found and some requirements, design techniques, and related trade-offs were extracted. In the subsequent rounds, the extracted requirements and design techniques were searched again to find original or additional resources explaining, repeating, or completing the same information. For scholarly discussions, forward snowballing process is also used [24] to examine the resources referenced in the retrieved resources.

The search and retrieval procedure of the resources has been initially performed in March to August 2018 using two search engines of Google and Google Scholar, and the same process has been repeated in August 2022 on the same search engines and five additional research databases as named above to update and extend the collected resources. The references retrieved in the second period are used to updated and strengthen the content and the references of the design knowledge already extracted in the first period. The report of the systematic review after the first period is available in [29].

2.1.4 Search Queries

To form the search queries two general steps are performed: First, a set of keywords are identified. Second, based on the identified keywords and their synonyms, a set of search strings are formed that can retrieve the most comprehensive collection of the related resources.

To retrieve the resources related to *RQ1* and to search for non-functional requirements, we have used two search queries: In the first query, we have used three main keywords of "API", "Non-Functional Requirement" and "Quality Attribute" (since in some resources non-functional requirements and quality attributes are used interchangeably) and have formed the following search strings:

```
{(Web API OR API) AND (Non-Functional Requirement* OR Quality Attribute* OR Quality*}
     OR {(Non-Functional Requirement*[2] OR Quality Attribute*) Of (WEB API OR API)}
```

The first query provided us with an initial set of resources mentioning some non-functional requirements. In the second query, we used the specific non-functional requirements obtained from the collected resources as keywords and combined them with "API" using the following pattern to form search strings. "API security" is an example of the following search pattern.

```
{(Web API OR API) AND (<Non-Functional Requirement>} OR {<Non-Functional Requirement>
                              Of (WEB API OR API)}
```

To retrieve the resources related to *RQ2* and to search for the available mechanisms, techniques, and patterns, we have used three queries. In the first query, we have used six keywords of "API", "Application Programming Interface", "Design", "Development", "Management", and "Architecture" and have formed the following search strings:

```
{(Web API OR API OR Application Programming Interface) AND (Design OR Develop* OR
                          Manag* OR Architect*)}
```

In the second query, we have used the specific non-functional requirements obtained from the previous search queries, and have looked for mechanisms, techniques, and patterns to address them. To this objective, we have used the six keywords of "Design", "Pattern", "Mechanism", "Technique", "API", and "Application Programming Interface" and have formed the following search strings. "API Security Pattern" is an example of the following search pattern.

```
{(Web API OR API or Application Programming Interface) AND <Non-Functional Require-
ment> AND (Design* OR Pattern* OR Mechanism* OR Technique*)} OR {(Design* OR Patterns*
OR Mechanism* OR Technique*) for (Web API OR API or Application Programming Interface)
                    AND <Non-Functional Requirement>}
```

In the third query, we have used the specific design techniques obtained from the two previous queries and have directly searched for the resources that contain them. "Open ID Connect" is an example of this query.

To retrieve the resources related to *RQ3* and to search for the evidence about the impact of the available techniques, mechanisms, and patterns on the identified non-functional requirements, we have used the nine keywords of "Analysis", "Evaluation", "Strength", "Weakness", "Effect", "Impact", "Trade-off", "advantage", and "disadvantage" and have formed the following search strings. "Analysis of Open ID Connect is an example".

```
{(analysis OR evaluation OR strength* OR weakness* OR impact* OR effect* OR advantage*
               OR disadvantage* OR trade-off*) of <Design Technique>}
```

2.1.5 Resource Selection and Inclusion

We have used five criteria to select from among scholarly articles and books and one criterion to select from among informal online expert discussions.

- *Criteria to Select Books and Scholarly Articles*: (i) The language of the source should be English. (ii) The resource should be sufficiently cited. To this objective, we have relied on Google Scholar search engine and

[2] The * symbol identifies variations of the search term, e.g., the plural form of a noun or gerund form of a verb.

selected the resources that appeared in the first four pages of the search results and were cited more than five times. To eliminate the bias against newly published papers, we have repeated the search process in the five popular databases named in Section 2.1.3 to retrieve resource published in reputable venues. (iii) The topic of the resource or its content should not focus on a specific programming language, or a specific implementation, or a specific product or platform. As a result of applying this criterion, we have eliminated the resources whose topic was about the design and implementation of APIs in a specific language such as Java, or merely focused on a specific implementation of APIs (such as REST APIs) excluding other implementations, or merely focused on explaining a specific product provided by an API vendor. (iv) The topic and content of resource should be related to or generalizable to Web APIs. As a result of this criterion, we have eliminated the resources whose content was only related to APIs other than Web APIs, such as programming languages APIs. (v) The resource should define, discuss, or evaluate one or more API non-functional requirements, or some concrete Web API architectural design techniques. As a result of applying this criterion, we have eliminated the resources whose topic and content was about the API development processes and activities or contained general guidelines that did not turn into a functional piece in API design.

- *Criteria to Select Weblogs, Tutorials, and White Papers*: (i) The online resource should be published by prominent API vendors or experts associated with these vendors. The prominent API vendors are identified based on [30]. Applying this criterion, we have eliminated webpages provided by question and answering platforms such as Stack Overflow, Information Technology (IT) publishing platforms, such as Medium.com, or IT bloggers or content providers such as TechTarget. Although, informal discussions have lower validity in comparison to books and scientific articles, we have not excluded these resources, but we have used them to increase the confidence in extracted pieces of design knowledge, i.e., if more resources are mentioning or repeating a non-functional requirement, a design technique, or a trade-off, it is more likely that the information is related and valid. (ii) The topic of the content should be general not explaining or advertising a specific platform or product.

Where several resources of different types have mentioned the same requirement, design technique, or trade-off, we have selected the most reliable and primary sources as the reference.

2.1.6 The Outcome of Resource Selection

The outcome of the collection procedure was 81 knowledge resources, including 7 books, 15 weblogs and tutorial, 5 vendor white papers, 6 design standards, and 47 research papers. (The collected resources are classified in Appendix I). As shown in Figure 1, from among the collected resources 59% were scholarly discussions (i.e., research papers) and 41% were expert discussions (i.e., design standards, books, weblogs and tutorials, and vendor white papers).

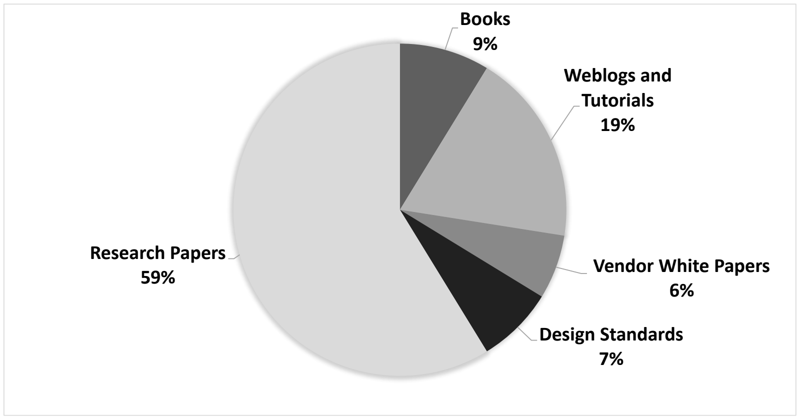


Figure 1 The Distribution of the Type of the Selected Knowledge Resources (N = 80)

2.2 Extracting Design Knowledge

To extract the pieces of design knowledge from the collected resources, we have used qualitative text analysis techniques [25][26][27] in two iterative steps: First, we have scanned the text of the selected resources to detect instances of non-functional requirements, design techniques, and the impact of design techniques on the non-functional requirements. Second, we have looked for the definition and other attributes of the identified requirements and techniques in the selected resources. To characterize the identified entities, we have used specific templates. The details of the extraction procedure are described in the following.

2.2.1 Extracting Non-Functional Requirements

To find instances of non-functional requirements, we have used the definition and categorization of non-functional requirements provided in the seminal work of [31]. [31] provides a comprehensive categorization of non-functional requirements: Non-functional requirements are requirements that describe how a software system will do its functions. Non-functional include different types of requirements, namely: (a) *quality attributes* of a software system (such as security, reliability, performance, availability), (b) *legal requirements* of a software system (such as privacy of end users and privacy of data), (c) *cost and budgetary requirements* of a system, (d) *implementation requirements* of a system (such as using a specific development environment or operating system), (e) *delivery requirements* of a system (time-to-release and time-to-market), and (f) *standard requirements* with which a system must follows (such as standard interfaces for interconnecting to external services). From the mentioned categories, we have focused on quality attributes and legal requirements. We have started from the mentioned instances as an initial list and have expanded the list while scanning the text of the selected resources.

To characterize the identified non-functional requirements, we have used a template containing the following information: (a) the name of the non-functional requirement, (b) the definition of the non-functional requirement, (c) the qualitative or quantitative metrics and measures based on which the non-functional requirement is evaluated (if available in the collected resources), and (d) the knowledge sources containing the information.

2.2.2 Extracting Design Techniques

To find instances of design techniques, we have looked for concrete functional fragments that are designed and implemented in the architecture of web APIs to address a specific non-functional requirement. To characterize the identified techniques, we have used a template close to the template of design patterns [32], a commonly used structure to share knowledge about software design techniques: containing the following information: (a) the name of the design technique, (b) the objective of the design; i.e., what functionality the design mechanism provides, (c) the description of design; i.e., a brief description of how the design works, (d) the model of the design; i.e., the class or sequence diagram of the design (if available), (e) the applicability condition of the design technique, explaining when and under what condition the design can be used (if available), and (f) the knowledge sources containing the information. The adopted template aids to unveil to the objective of the design techniques and provides an initial structure that facilitates relating design techniques to non-functional requirements.

2.2.3 Extracting Trade-Offs

To identify instances of the trade-offs, we have looked for the analysis or evaluation of the identified techniques in the collected resources. To this objective, we have scanned the parts of the text that explicitly mention the keywords that refer to trade-off, namely “advantage”, “disadvantage”, “strength”, “weakness”, “effect”, “side-effect”, “trade-off”, “evaluation”, or “analysis”. We have also read the text to find implicit discussions that could not be detected using the above keywords. To extract and analyze the impact of the identified design techniques on non-functional requirements, we have used a template containing following information: (a) the name of the technique, (b) the related or affected non-functional requirement, (c) justification and evidence for the identified trade-off, and (d) the knowledge sources containing the information.

2.3 Aggregating and Classifying Design Knowledge

We have aggregated and categorized the extracted pieces of design knowledge as described in the following.

2.3.1 Aggregating and Classifying the Knowledge Related to Non-Functional Requirements

To categorize the identified non-functional requirements, we have performed five steps: (1) We have used ISO / IEC 25011 standard [33] for system and software quality requirements and evaluation to check and define the non-functional requirements (if defined in the standard), and to classify and group similar quality requirements. ISO / IEC 25011 standard identifies a general list of software quality requirements and defines them. We have used this general list and its definitions as a reference and have customized and extended it to the specific domain of APIs. Accordingly, in some cases, a quality requirement (i.e., non-functional requirement) as defined by IEC has been different from the definition that has been applicable to an API. In these cases, we have kept the definition related to the domain of APIs. (2) In cases where various resources have characterized a requirement differently, we have considered the commonalities between these characterizations. (3) We have merged non-functional requirements that have the same definition but are referred to with different terms in different resources, and we have named them with one of the terms. (4) We have grouped similar requirements into one category based on the concept that they were referring to. (5) We have broken down a non-functional requirement into sub-types and sub-categories if the sub-types have their own definition and specific metrics or evaluations are performed to assess them.

2.3.2 Aggregating and Classifying the Knowledge Related to Design Techniques

To categorize the extracted design techniques, we have performed the following steps: (1) We have focused on the architecture of design techniques; i.e., the components and the interactions between them at the logical level [34], [35]; i.e. we have abstracted away the details of the techniques that depend on development methodology (such as micro-service or service-oriented approach) or the choices of implementation and communication protocol (such as Representational State Transfer (REST) APIs or Simple Object Access Protocol (SOAP) APIs) and have exposed the high-level design of the extracted mechanisms. (2) We have merged the design techniques and patterns that implement the same approach but are referred to with different terms. (3) We have categorized the design techniques based on their objectives, i.e., the functionality that the techniques provide or the non-functional requirement that they address. (4) We have related the objectives of the techniques to the non-functional requirement that they address.

2.3.3 Aggregating and Classifying the Knowledge Related to Trade-Offs

To categorize the identified trade-offs, we have used two criteria: (a) the type and extent of the effect of a design mechanism on a non-functional requirement, and (b) the type of evidence. The type of an effect is categorized as positive (represented by "+" label) or negative (represented by "–" label). The extent of the effect is categorized as "*weak*", "*medium*", "*to some extent*", and "*strong*". The type and strength of an effect of a design mechanism on a non-functional requirement are identified based on analyzing the reasons and argumentation provided by an expert, or by analyzing the mechanism against the design guidelines or principles that experts suggest, or by empirical evidence extracted from the collected resources. "*weak*" and "*strong*" labels are used when there is a clue or evidence that the effect of a technique on a non-functional requirement is strong or weak. A "*medium*" label is assigned when there have been several alternatives addressing the same design objective, and these alternatives could be compared to each other in terms of the strength of their effect, or where their effect is neither weak, nor strong. "*To some extent*" impact has been used when the extent of the impact could not be identified because of varying evidence or arguments, or when the impact has been neither weak, nor strong. The type of evidence for the identified effects is categorized into three level of strengths: (a) support with sound reasoning and justification (based on comparison to some design principles), (b) support with expert opinion and anecdotal evidence, and (c) support with empirical and scientific evidence or formal proofs. The type of evidence is identified according to the type of argument made to justify the type and extent of an effect.

2.4 Summarizing and Visualizing Design Knowledge

We have summarized and visualized the structured knowledge using a kind of knowledge graphs [28]. The knowledge graphs are directed and typed and illustrate the relationships between the identified non-functional requirements and design techniques. As shown in Figure 2, the nodes in the presented knowledge graphs represent three entities: (a) non-functional requirements (labeled with NFR), (b) design objectives (labeled with DES-OBJ), or (c) design technique (labeled with DES-TECH). These entities are identified based on the "*Non-Functional Requirement*" field of the template used to extract information related to non-functional requirements, and the "*Design Technique*", and

“*Design Objective*” fields of the template used to extract and organize information related to design techniques. The edges are directed and typed and represent directed relationships between entities. The edges have one of the following types: “*IS-A*”, “*REALIZES*”, and “*EFFECTS*”. The “*IS-A*” relationship shows the hierarchical decomposition relationship between two non-functional requirements or between two design objectives. The “*REALIZES*” relationship shows that a design objective addresses a non-functional requirement, or a design technique implements a design objective. The “*EFFECTS*” relationship shows the type and extent of the effect of a design mechanism on the related non-functional requirements and has one of the following labels: “Strong+”, “Some+”, “Weak+”, “Strong–”, “Some–”, “Weak–”.

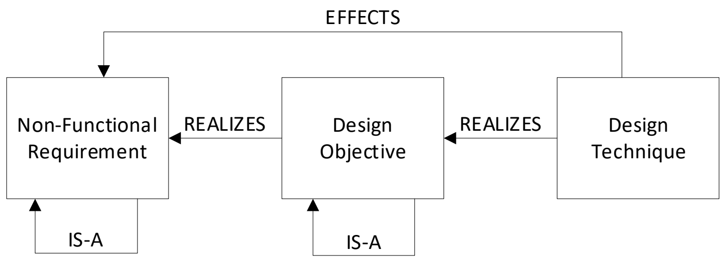


Figure 2 Entities and Relationships in the Presented Knowledge Graphs

3 Non-Functional Requirements of APIs

We extracted and structured 9 categories and 27 distinct non-functional requirements for web APIs in the body of WEBAPIK as reviewed in the following.

3.1 Adoptability

Adoptability of an API is the extent to which an API is adopted or can be potentially adopted by the developers to be used in their code. API adoptability can be evaluated by measures such as the number of API users (i.e., developers using an API) or the number of different third-party applications and services in which the API is used [8], [36].

3.2 Visibility

Visibility of an API is the extent to which an API is visible to the potential clients and users, and the ease with which an API can be discovered by the users. Visibility of an API can be evaluated by criteria such as the type of clients and the users to which the API is visible. Based on the types of clients, the visibility of an API can be categorized into private, protected, or public or open [11], [19].

3.3 Accessibility

Accessibility of an API is the degree of ease with which an API can be accessed by client applications and services. Accessibility of an API can be broken down into access simplicity, access duration, and access rate [8], [11], [37] (Figure 3).

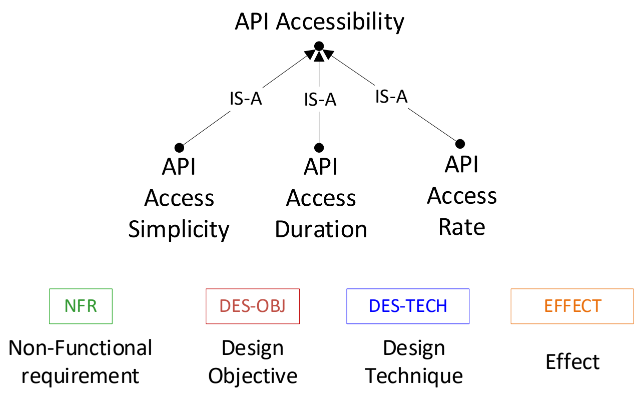


Figure 3 API Accessibility Requirements

NFR Access Simplicity: Access simplicity identifies how easily an API can be accessed by the clients. The ease of access to an API can be evaluated by criteria such as: (a) the number of interactions and security checks between a client and an API before the client can access the API, (b) the number of actors involved in the access permission process, (c) the time it takes for a client to obtain access to an API, and (d) the limitations and constraints that clients have to access an API.

NFR Access Duration: The duration of access identifies the time frame in which a client can access to an API; i.e. whether upon permission, the client can access the API once or several times and whether the access is given for a short or a long period of time.

NFR Access Rate: Access rate of an API (or access frequency) identifies how many times an API can be accessed by a client over a specific period upon permission.

3.4 Evolvability

Evolvability of an API is the degree of ease with which an API (i.e., API message parameters, API calls, or the behaviour of an API) can be modified, upgraded, or changed over time without affecting its clients. An API may be used in numerous applications and services. Therefore, any change to an API affects all the clients that are using it and may require the client code to change. Changes to an API can be minor or major (Figure 4). New versions of APIs should be compatible with previous versions (i.e., be backward compatible) in the face of both minor and major changes. Evolvability is specifically important for public APIs, which are used in a wide range of third-party applications and services [11], [38], [39].

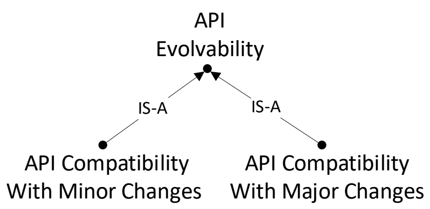

API Evolvability
IS-A
IS-A
API Compatibility With Minor Changes
API Compatibility With Major Changes

Figure 4 API Evolvability Requirements

NFR API Compatibility with Minor Changes: Changes to an API can be minor, such as changes to the message parameters and the API calls. The API should accommodate these kinds of changes without requiring the clients to change.

NFR API Compatibility with Major Changes: Changes to an API can be major, such as changes to the behaviour and services of the API. The API should accommodate these kinds of changes while minimizing the changes required in the client-side.

3.5 Usability

Usability of an API (also referred to as Developer Experience (DevX)) is the degree of ease with which developers can use the API in their code to achieve their development goals. An API should be easily understood and learned by the developers to be used in their code. Usability of an API can be broken down into understandability, efficiency, usage simplicity and consistency (Figure 5) [8], [9], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51].

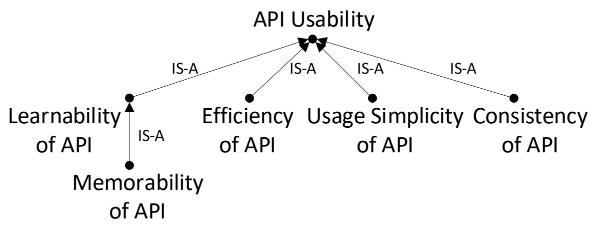


Figure 5 API Usability Requirements

NFR Learnability: Learnability of an API (also referred to as Understandability of an API) is the degree of ease with the operation and usage of the API can be understood and learned by developers. Quality attributes such as memorability an API contribute to the learnability of the API.

NFR Memorability: How easily the API calls can be remembered.

NFR Efficiency: How efficiently the API can be used by developers for various development tasks.

NFR Usage Simplicity: How easy it is to use an API in a client code, and what kinds of limitations and constraints exist for using an API.

NFR Consistency: How obviously the API purpose can be inferred from the API contract and how well an API matches the developers' mental model.

3.6 Performance

Performance of an API is the extent to which an API can respond to the requests of clients in a prompt and timely manner. The performance of an API can be broken down into response time, latency, throughput, and availability (Figure 6) [11], [37], [52].

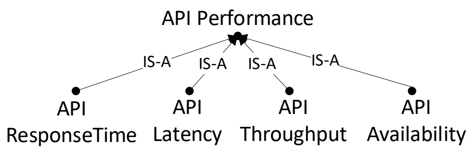


Figure 6 API Performance Requirements

NFR Response Time: Response time of an API is the time it takes for an API to process a request and to respond to the request. The response time of an API is measured in terms of the difference between the time at which a client's request is received by an API and the time at which a response is sent to a client.

NFR Latency: Latency of an API (also referred to as end-user latency of an API) is the delay that a client perceives between sending a request and receiving the response of an API. API latency is measured as the difference between the time at which a request is submitted by a client to an API and the time at which the response is received by the client. Latency is specifically important for web APIs since network delays are of the order of millisecond or higher.

NFR Throughput: Throughput of an API (also referred to as capacity of an API) is the degree to which the request load of an API can increase without affecting the behavior of the API towards the clients. The throughput of an API can be measured as the maximum number of requests that can be processed by the API per unit of time without any failure in operation or without affecting the behavior of the API.

NFR Availability: Availability of an API (also referred to as up-time) is the degree to which an API provides continuous service over time. Availability is important for the APIs that provide time-critical services to their clients. Availability of an API can be measured as the amount of time that an API is ready for service over a specific period of time.

3.7 Extensibility

Extensibility of an API is the degree of each with which the clients and the backend services of an API can change at runtime and is comprised of two requirements (Figure 7): (1) client-side extensibility: The degree of ease with which different clients can be added to and removed from an API at run-time, and use an API without modifying the API or the back-end services; and (2) server-side extensibility: the degree of ease with which different back-end services can be added to or removed from an API at run-time without modifying the API or the clients [11].

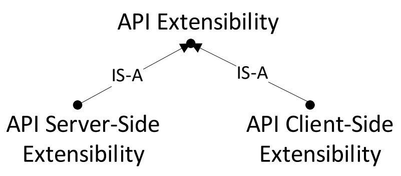


Figure 7 API Extensibility Requirements

3.8 Interoperability

Interoperability of an API (also referred to as adaptability, flexibility, or Interface compatibility) is the degree to which an API can work with different types of clients that have various communication formats. Interoperability of an API can be broken down into flexibility towards different message formats, flexibility towards different message parameters, and flexibility towards different communications protocols (Figure 8) [39].

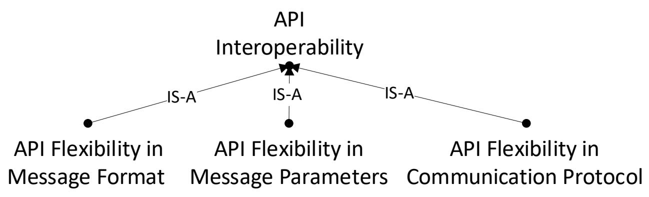

API Interoperability
IS-A
IS-A
IS-A
API Flexibility in Message Format
API Flexibility in Message Parameters
API Flexibility in Communication Protocol

Figure 8 API Interoperability Requirements

3.9 Security

Security of an API is the degree to which an API is safe against unauthorized or unintended access, unintended or unauthorized message disclosure, corruption, and modification, as well as service failures. APIs expose back-end systems and data to external applications and services. This exposure increases the risk of malicious attacks to the back-end systems. This risk is specifically high in open APIs, which are accessible to a wide range of clients. Some attacks that can be made to an API are as follows:

- *Denial of service attacks:* Malicious clients might attack the back-end systems by making a large number of calls to an API. This kind of attack prevents other clients from accessing the back-end system and may shut down the back-end systems.
- *Man-in-the-middle attacks*: A malicious attacker sits in between a client and API and corrupts and alters the requests and responses between an API and its clients.
- *Eavesdropping:* Malicious clients and attackers may secretly listen to the communications between an API and client without their consent and permission.

Security of an API can be broken down into confidentiality, privacy, operational security, and reliability (Figure 9) [10], [11], [12], [38], [39], [53], [54], [13].

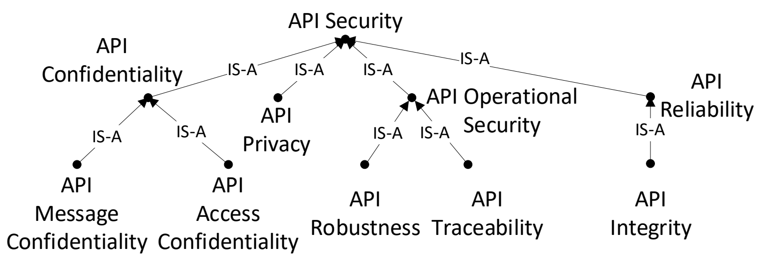
(1) 

Figure 9 API Security Requirements

3.9.1 Confidentiality

Confidentiality of an API is the degree to which an API services and messages are protected against being discovered, observed, and accessed by unintended or unauthorized audiences. API confidentiality can be broken down into (a) message confidentiality, and (b) service confidentiality.

NFR Message Confidentiality: Message confidentiality of an API is the degree to which the requests and responses of an API are protected against being observed and accessed by unintended and malicious audiences.

NFR Access Confidentiality: Access confidentiality is the degree to which the provided data and services of an API are protected against unauthorized access and usage by unauthorized clients.

3.9.2 Privacy

Privacy of an API is the degree to which the rights of involved human parties are preserved upon access to and receiving service from the API. An API should have both service privacy and access privacy. APIs may expose sensitive data to their client application and services. This data may belong to the API provider or to an end user. The related end-users and human parties should have control over granting and revoking permission to access an API. Moreover, APIs may receive sensitive data from their client application and services. This data may belong to the clients or other human parties. An API should also preserve the privacy of the received sensitive data and protect it from unintended and unauthorized observation.

3.9.3 Operational Security

Operational security of an API is the degree to which the API performs its expected operations safely and correctly in the face of problems and failures. Operational security can be broken down into robustness and traceability.

NFR Robustness: Robustness of an API is the degree to which an API can tolerate failures in the back-end systems or attacks of malicious clients, and continue safe, stable, and healthy operation over time.

NFR Traceability: Traceability of an API is the degree to which an API keeps the record and the trace of its requests and its responses.

3.9.4 Reliability

Reliability of an API (also referred to as trustworthiness of an API) is the degree to which an API behaves in conformance with what is expected from it, and the extent to which the clients can trust and rely on the services that the

API provides. For example, to what extent the API ensures that it will respond to the clients' requests, to what extent the API will respond to the clients' requests within a specified time limit, and to what extent it will provide accurate responses upon which the clients can trust. Reliability of an API includes requirements such as integrity.

NFR Integrity: Integrity of an API is the degree to which an API can ensure that it does not drop, ignore, or corrupt the requests of the clients and the responses of the back-end services.

4 Design Techniques to Address Non-Functional Requirements in APIs

We extracted and structured 43 high-level design objectives and 37 concrete design techniques in the body of WEBAPIK as reviewed in the following.

4.1 Mechanisms to Address Evolvability in APIs

APIs need to evolve over time to upgrade the offered services or to address new requirements. API version management mechanisms are specifically designed to address the evolution of APIs over time. API version management mechanisms minimize the impact of these continuous changes on the API clients and can be divided into two groups (Figure 10): a) mechanisms to accommodate minor changes, and b) mechanisms to address major changes to APIs [11], [39].

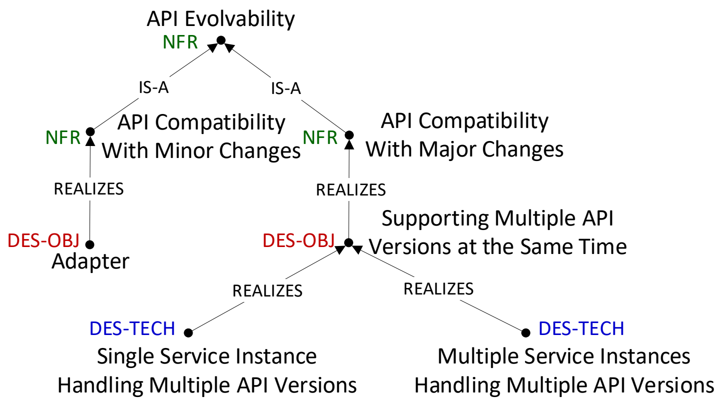

API Evolvability
NFR
IS-A
IS-A
NFR
API Compatibility With Minor Changes
NFR
API Compatibility With Major Changes
REALIZES
REALIZES
DES-OBJ
Adapter
DES-OBJ
Supporting Multiple API Versions at the Same Time
REALIZES
REALIZES
DES-TECH
Single Service Instance Handling Multiple API Versions
DES-TECH
Multiple Service Instances Handling Multiple API Versions

Figure 10 API Evolvability Design Techniques

DES-OBJ Adapter: To handle minor changes in APIs, adaptors are designed that adapt old interfaces to new ones.

DES-OBJ Supporting Multiple API Versions at the Same Time: To support major changes, specific mechanisms need to be designed that can accommodate multiple versions of API at the same time and notify the clients about retiring an API. Two design alternatives for supporting different versions of an APIs are described in the following.

DES-TECH Single Service Instance Handling Multiple API Versions: Multiple versions of an API is supported by a single instance of the back-end service.

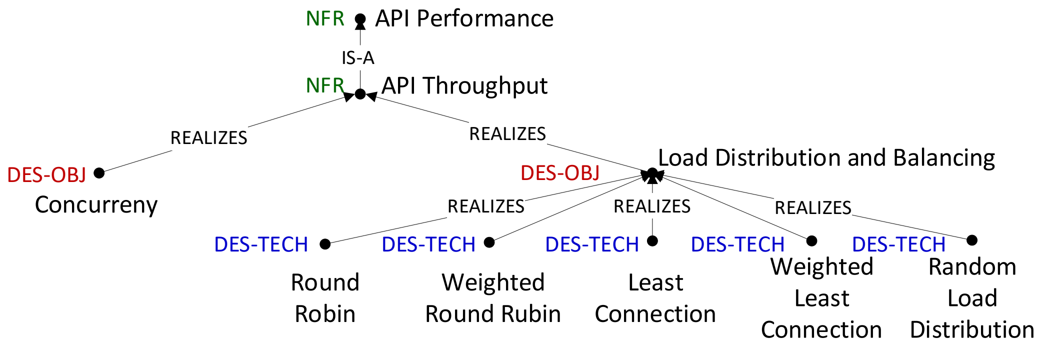
DES-TECH Multiple Service Instances Handling Multiple API Versions: Different instances of the back-end services are designed to support each version of an API.

4.2 Mechanisms to Address Performance in APIs

Three groups of mechanisms address performance requirements in APIs, namely: (1) mechanisms to address throughput, (2) mechanisms to address response time, and (3) mechanisms to address availability in APIs [11], [12], [13], [56], [57], [58].

4.2.1 Mechanisms to Address Throughput in APIs

Throughput mechanisms allow an API to manage high loads of API calls gracefully. Two specific mechanisms designed to increase the throughput of APIs include concurrency, and load distribution and balancing (Figure 11).



NFR API Performance
IS-A
NFR API Throughput
REALIZES
REALIZES
DES-OBJ Concurreny
DES-OBJ Load Distribution and Balancing
REALIZES
REALIZES
REALIZES
DES-TECH Round Robin
DES-TECH Weighted Round Rubin
DES-TECH Least Connection
DES-TECH Weighted Least Connection
DES-TECH Random Load Distribution

Figure 11 API Throughput Design Techniques

DES-OBJ Concurrency: Concurrency mechanisms (also referred to as load concurrency) allow to run the same instance of a service on multiple backend systems so that an API can handle multiple clients simultaneously.

DES-OBJ Load Distribution and Balancing: Load distribution and balancing mechanisms are responsible for distributing clients' requests evenly between the backend servers offering the same service. The requests are distributed between the backend servers using a load distribution strategy, including Round Robin, Weighted Round Robin, Least Connection, Weighted Least Connection, Random. The distribution strategies are described in the following.

DES-TECH Round Robin: In the round robin strategy, the requests are assigned to the backend servers offering the same service in a sequential and cyclic order. This strategy is used when backend systems have similar capacity and configuration.

DES-TECH Weighted Round Robin: In the weighted round robin strategy, a weight or rate is assigned to each server based on the capacity of that server. The number of requests sent to a server in each round is determined based on the assigned weight.

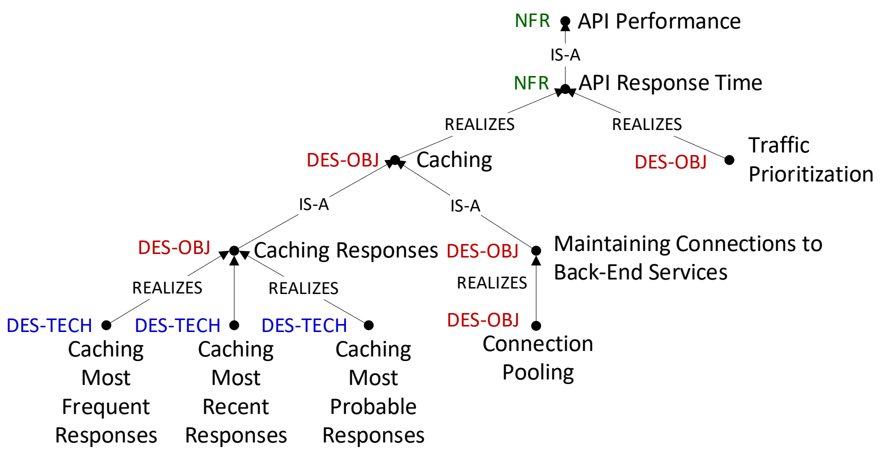
DES-TECH Least Connection: In the least connection technique, the requests are sent to the backend servers according to their current load. The server with the lowest request load will receive the next request.

DES-TECH Weighted Least Connection: In weighted least connection strategy, a weight is assigned to each backend server based on the capacity of the server. Requests are then assigned to the backend servers based on their level of load and their weights.

DES-TECH Random Load Distribution: In random load distribution, requests are assigned randomly to the backend servers offering the same service. This strategy is used when backend systems have similar capacity and configuration.

4.2.2 Mechanisms to Address Response Time in APIs

Two specific group of mechanisms designed to control and reduce the response time of an API include caching and traffic prioritization mechanisms (Figure 12)



NFR API Performance
IS-A
NFR API Response Time
REALIZES
REALIZES
DES-OBJ Caching
DES-OBJ Traffic Prioritization
IS-A
IS-A
DES-OBJ Caching Responses
DES-OBJ Maintaining Connections to Back-End Services
REALIZES
REALIZES
REALIZES
DES-TECH Caching Most Frequent Responses
DES-TECH Caching Most Recent Responses
DES-TECH Caching Most Probable Responses
DES-OBJ Connection Pooling

Figure 12 API Response Time Design Techniques

DES-OBJ Caching: The objective of caching mechanisms is to store and maintain the interactions and connections with the backend services to use them for future requests. Caching mechanisms can be divided into two groups: caching the responses of APIs and maintaining the connections to the backend systems.

DES-OBJ Caching API Responses: In these mechanisms, the previous responses of the backend services to specific requests are stored in a cache memory closer to the clients than the backend services. These in-memory responses are then used to respond to subsequent clients without referring to the back-end services. Mechanisms should also be in designed in a cache that ensure the stored responses are up to date. Caching mechanisms are specifically used when the responses of the back-end service do not change frequently or when the response changes periodically. Three concrete strategies to store the responses of the backend services include storing the responses to the most frequent requests, the responses to the most recent requests, and the responses to the requests that will frequently occur in the future.

DES-TECH Caching Most Frequent Responses: In this strategy, the responses to the most frequent requests are stored in the cache memory.

DES-TECH Caching Most Recent Responses: In this strategy, the responses to the most recent requests are stored in the cache memory.

DES-TECH Caching Most Probable Responses: In this strategy, the future requests of the clients are predicted based on the history of previous requests and the responses to the requests are stored in the cache memory.

DES-OBJ Maintaining Connections to Back-End Systems: Finding and connecting to the back-end services that respond to the clients' requests is time consuming. To reduce the response time, once these connections are established, they are kept alive for similar future requests. One concrete technique to maintain connections to the backend systems is connection pooling.

DES-OBJ Connection Pooling: Connection pooling allows to keep the connections to the back-end services and databases active. Once a connection is established, it is placed in the pool of connections for future reuse. This mechanism eliminates the time for setting up a connection to the backend services in responding to each client's request.

DES-OBJ Traffic Prioritization: The objective of traffic prioritization mechanisms is to control the order in which the requests of clients are responded by an API. For this purpose, a priority is assigned to each client. The order in which the API calls are responded are determined based on the assigned priorities. Priority queues need to be in place to group the request of different clients.

4.2.3 Mechanisms to Address Availability in APIs

Redundancy and Replication mechanisms are specifically designed to help APIs provide service continuously over time even in the face of failure (Figure 13).

DES-OBJ Back-End Service Replication: In replication and redundancy mechanisms, the same instance of a service is duplicated on multiple back-end servers so that if one server fails, others can be replaced to continue service provision.

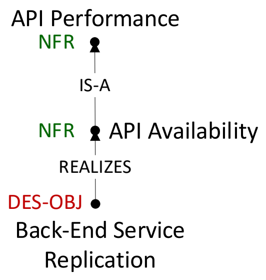


Figure 13 API Availability Design Techniques

4.3 Mechanisms to Address Extensibility in APIs

The systems and services behind an API change over time. Mechanisms are required that allow to add new back-end services and remove old ones without affecting the run-time behaviour of an API and its clients. Moreover, the clients of an API change over time. Mechanisms need also to be in place that allow to add or remove clients without requiring any run-time changes to the API or the backend systems. Some mechanisms specifically designed to address server-side extensibility in APIs include API gateway, service registration, service discovery, service mapping and composition, and service orchestration (Figure 14)[11], [12], [14], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68] [69].

DES-OBJ API Gateway: API gateway (also referred to as gateway or API facade) is a logical or physical component that sits between backend systems and the clients, encapsulates the internal structure and behaviour of the backend systems, and act as a single access point to the back-end systems. API gateway also handles all the communications and interactions that happen between backend systems and clients. All requests submitted by the clients first pass through the gateway to reach to the backend. An API gateway can have various responsibilities, including: (1) creating and managing APIs, (2) customizing APIs for different clients, (3) authorizing the clients, (4) translating API requests and the responses, and (5) monitoring and managing the traffic of requests. API gateway is especially useful in opening legacy systems and platforms which are often heterogeneous and are not expected to change. An API gateway allows to interconnect legacy systems with external applications and services while the minimum changes to the backend. Two alternative techniques to design an API gateway are central gateway and back-end for front-end.

DES-TECH Central Gateway: Central gateway (also referred to as single gateway) is one physical or logical component that handles all the requests of all types of clients.

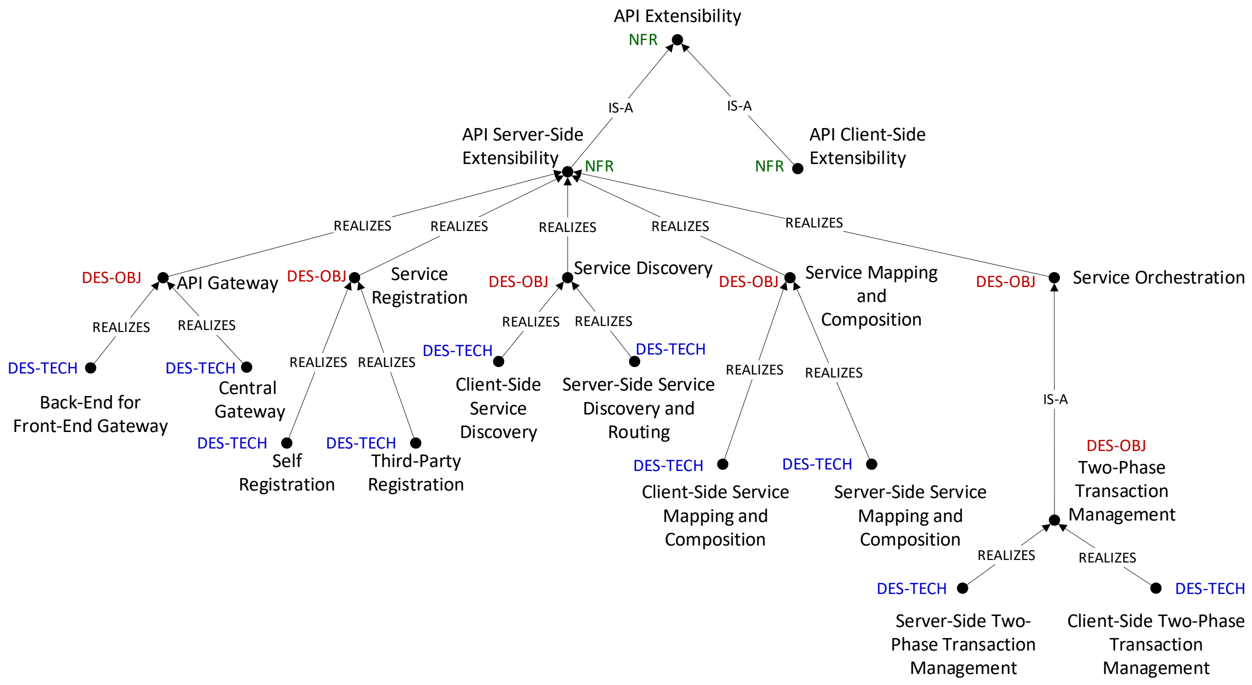


Figure 14 API Extensibility Design Techniques

DES-TECH Backend for Frontend Gateway: In Backend for frontend gateway, multiple gateways are designed each of which handles the requests of a specific type of client.

DES-OBJ Service Registration: The objective of service registration mechanism is to handle dynamic addition and removal of backend services. Service registration is composed of a service registry component (i.e.; a look-up table or database) where new instances of back-end services and objects register and de-register themselves as a service provider. There are two alternatives for registering the back-end services: self-registration, and third-party registration.

DES-TECH Self-Registration: In self-registration, each new instance of a service is responsible for registering and deregistering itself in the service registry. The service instances are also responsible for sending heartbeat requests to the service registry to refresh their registration.

DES-TECH Third-Party Registration: In third-party registration, a third-party registrar (or the API provider) is responsible for updating the service registry and registering and de-registering instances of backend services. The service registrar should continuously monitor changes to the set of running back-end services to detect addition and removal of service instances.

DES-OBJ Service Discovery: The objective of service discovery mechanism is to find the instances of backend services that are related to a client request. To this objective, a service registry component (i.e.; a look-up table or database) should be in place where instances of back-end services and objects can register and de-register themselves into it. Moreover, the clients or an intermediary should be able to look up the registry to find out the related services. This technique is specially used when the back-end services or their location (physical or logical address) change dynamically. There are two alternatives for designing the service discovery mechanism: server-side discovery and client-side discovery.

DES-TECH Server-Side Service Discovery: In server-side discovery, the API provider or a server-side router is responsible for querying the service registry, load balancing, selecting an available instance of the service, and routing the request to the related instance.

DES-TECH Client-Side Service Discovery: In client-side service discovery, the client is responsible for querying the service registry, using a load balancing mechanism to select an available service instance, selecting an available instance, and sending a request.

DES-OBJ Service Composition and Mapping: The objective of API composition and mapping mechanisms is to translate and map the request of a client onto the related backend services, to call the related backend services in an appropriate order, and to return the response to the client. The API composer can call out the back-end services either sequentially or in parallel. Two alternative techniques to design API composition and mapping include: server-side API composition and client-side API composition.

DES-TECH Server-Side API Composition: In server-side API composition, the API provider is responsible for mapping the clients' requests onto the backend services and calling the related backend services in appropriate order. This technique provides a single and simple interface for a complex backend.

DES-TECH Client-Side API Composition: In client-side API composition, the client is responsible for mapping and calling the backend services related to a request.

DES-OBJ Service Orchestration: Sometimes to respond to a client's request several back-end services should collaborate. Service orchestration mechanisms are responsible for preparing and coordinating the related back-end services. One specific instance of a service orchestration mechanism is two-phase transaction management.

DES-OBJ Two-Phase Transaction Management: The objective of two-phase transaction mechanism (also referred to as two-phase commit) is to prepare and coordinate the related back-end services before responding to a client's request. To this objective, the API provider or the client first checks the availability of the related back-ends services, prepares the backend services to perform the related operation, sends the request of the client to them, and finally receives the response. Two alternative techniques to design two-phase transaction management are server-side transaction management and client-side transaction management.

DES-TECH Server-Side Transaction Management: In this technique, the API provider is responsible for preparing and coordinating the related back-end services before responding to a request.

DES-TECH Client-Side Transaction Management: In this technique, clients are responsible for coordinating and preparing the backend services before sending the request to the backend.

4.4 Mechanisms to Address Interoperability in APIs

An API may have different clients. These clients may work with different communication protocols and styles and may have different message formats or different message parameters. Interface translation mechanisms (including message format conversion, protocol translation, and adapters), and communication styles are particularly designed to allow an API to interact with different kinds of clients (Figure 15) [39], [65], [70], [71].

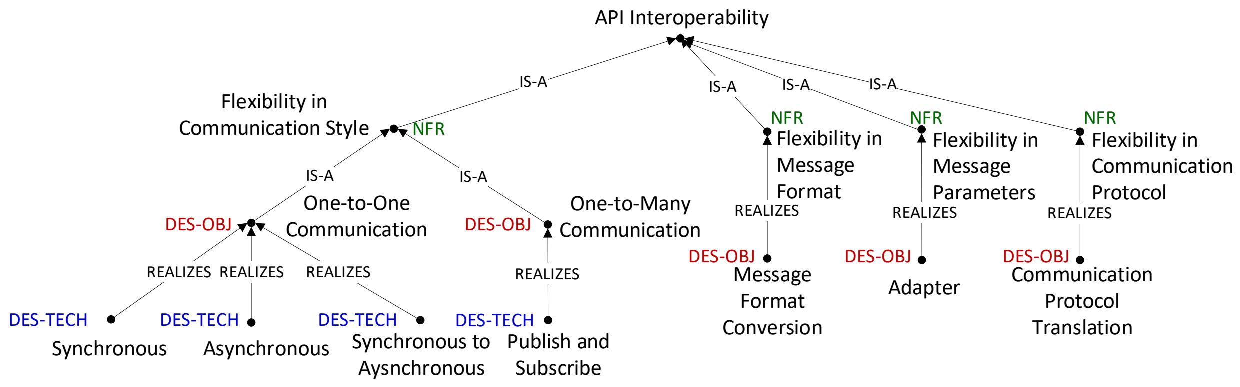


Figure 15 API Interoperability Design Techniques

DES-OBJ Message Format Conversion: Clients may have a different message format from an API. The objective of message format conversion mechanisms is to convert the message format of the clients and the API to one another (for example, changing the message format of JSON to XML or HTML or vice versa).

DES-OBJ Adapter: The clients may have different message parameters from the backend services. Adapters translate the requests of the clients into API calls for back-end services.

DES-OBJ Communication Protocol Translation: The clients many have different communication protocol from an API. Mechanisms need to be in place to translate different communication protocols to each other (for example, changing Simple Object Access Protocol (SOAP) protocol to representational state transfer (REST) protocol and vice versa).

DES-OBJ Communication Styles: API interaction and communication styles identify the way that an API communicate with its clients. These styles can be categorized into two groups of one-to-one and one-to-many styles.

DES-OBJ One-To-One Communication Style: In one-to-one communication, an API interacts with one client at a time to receive and respond to the client's request.

DES-TECH Synchronous Communication: The objective of synchronous communication mechanisms is to respond to the request of clients immediately. To this objective, a time limit is specified for responding to a client's request. Upon submitting a request, the client expects to receive the response with the specified time limit. Synchronous communication mechanism is specifically used for critical requests and those request that need to be responded in real-time. The API provider has two alternatives in responding to the synchronous communications: (1) It may block its other requests in order to respond to a single request; or (2) it may attach a high-priority to the request and put is in a high-priority queue to be processed as the next request. The client as well has two alternatives until it receives the response of the API: (1) The client may block and wait until the response is received, or (2) It may continue its operation until the response is received.

DES-TECH Asynchronous Communication: The objective of asynchronous communication mechanism is to respond to the clients' requests in a non-blocking manner. To this objective, the API provider attaches a priority to a client's request upon receiving the request and put it in a waiting-list queue. When the response is ready, the API notifies the client and send the response back to the client. In this technique, neither the client nor the API provider block for the communication. There are various alternatives for designing the waiting list queue, such as First-In First Out queues (FIFO) or Priority Queues. There are also two variants for sending back the response for the client: (1) Calling Back the client: The API calls backs the client with the response. (2) Polling for response: The client polls for the response. This alternative requires a response queue in addition to the request queues.

DES-TECH Synchronous to Asynchronous Communication: The objective of synchronous to asynchronous technique is to respond back to a client's request in a timely and reliable manner. The synchronous to a synchronous mechanism is specially used when the client expect a synchronous response, but the back-end systems provide asynchronous response. For this purpose, the API facade or gateway acts as an intermediary and allows the clients to communicate synchronously while it communicates with the back-end systems a synchronously. The API gateway may need to poll for the response of the back-end system until the response is received.

DES-OBJ One-To-Many Communication Style: In one-to-many style, an API interacts with a group of clients that have the same request at a time and respond to their request in group. One specific technique to design one-to-many communication style is Publish and Subscribe.

DES-TECH Publish and Subscribe: The objective of Publish and Subscribe mechanism is to send the same response to several clients or to send the same request to several instances of back-end services. This mechanism is mainly used to notify a set of clients or backend services of an update. To receive the messages, the clients as well as the back-end services subscribe to (or unsubscribe from) an intermediary which is responsible for informing the clients and the back-end services of the new updates. There are two main alternatives for subscription: (1) Topic-based subscription in which the receivers declare their interest in receiving messages about a topic or (2) Content-based subscription in which the receivers declare their interest in receiving a specific content. The intermediary sends the message to the subscribers if the content or some parts of it matches the subscribers' constraints. There are also two implementations for informing the subscribers: (1) Pushing the messages to the recipients. In this case, messages can be customized for each subscriber; or (2) Pulling the messages in which recipients pull the message from the intermediary.

4.5 Mechanisms to Address Security in APIs

Web APIs often provide sensitive services and data to their clients and expose the back-end systems of a platform to the clients. Mechanisms need to be designed that can control and protect the security of the APIs and the back-end systems against unauthorized and malicious clients. Security mechanisms can be divided into four categories: (a) mechanisms to address confidentiality in web APIs, (b) mechanism to address privacy in web APIs, (c) mechanism to address robustness in web APIs, and (d) mechanisms to address traceability in web API interactions [10], [11], [12], [13], [15], [16], [39], [55], [72], [73][3], [74], [75], [76], [78], [79], [80], [81], [82].

4.5.1 Mechanisms to Address Confidentiality in APIs

Confidentiality mechanisms are responsible for ensuring that the requests and response of an API are only visible and accessible by authorized clients and audience. These mechanisms can be divided into two categories: (a) techniques to address access confidentiality and (b) techniques to address message confidentiality (Figure 16).

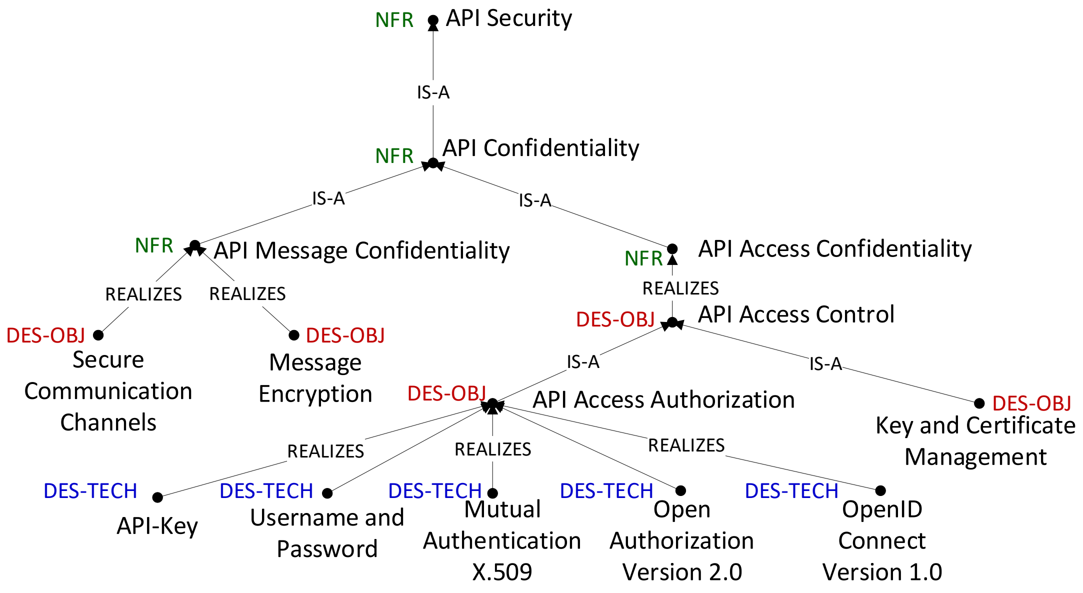

NFR API Security
IS-A
NFR API Confidentiality
IS-A
IS-A
NFR API Message Confidentiality
NFR API Access Confidentiality
REALIZES
REALIZES
REALIZES
DES-OBJ Secure Communication Channels
DES-OBJ Message Encryption
DES-OBJ API Access Control
IS-A
IS-A
DES-OBJ API Access Authorization
DES-OBJ Key and Certificate Management
REALIZES
REALIZES
REALIZES
DES-TECH API-Key
DES-TECH Username and Password
DES-TECH Mutual Authentication X.509
DES-TECH Open Authorization Version 2.0
DES-TECH OpenID Connect Version 1.0

Figure 16 API Confidentiality Design Techniques

DES-OBJ Access Control: Access control mechanisms are responsible for identifying and authenticating clients and authorizing their access to the API. These mechanisms can be divided into two groups: (1) key and certificate management, and (b) access authorization.

DES-OBJ Key and Certificate Management: Key and certificate management mechanisms are responsible for managing and validating the keys and certificates that clients provide to access an API.

[3] X .509 is proposed and implemented for communications over internet, and with HTML and REST APIs. However, we expose and focus on the general mutual authentication pattern used in this mechanism.

DES-OBJ Access Authorization: Access authorization mechanisms are responsible for verifying the clients and permitting access to an API. Authorization responsibilities includes controlling the APIs that a client can view and access, controlling the level of access of a client to an API and the methods that it can invoke. Some concrete design techniques to authorize access to APIs include API-Key, Username and Password, Mutual and Certificate-Based Authentication, Open-Authorization Version 2.0 and Open-ID Connect Version 1.0.

DES-TECH API-Key: The objective of API-Key (also referred to as Client-ID, App-Key, App-ID or Consumer-ID) is to identify the client applications and services that use an API. For this purpose, the API provider issues a unique key (referred to as API-key) for each client application and services that wants to access an API. The API-key is an alphanumeric string that is issued upon the registration of a client.

DES-TECH Username and Password: The objective of username and password mechanism is to authorize the access of an end-user who want to use an API via a client application or service. For this purpose, the client presents unique credentials (i.e.; the username and password) to the API provider. The API provider validates the received credentials against its credential store and authorizes the access. The client's credentials are encoded by the client before transmission to the API and need to be decoded by the API. Username and Password technique can be used in combination with API-key for achieving a higher level of security

DES-TECH Mutual and Certificate-Based Authentication: The objective of Mutual and Certificate-Based Authentication (X.509) (also referred to as Two-Way Authentication and Two-Way Handshake) is to authenticate both parties that want to communicate with each other. For this purpose, both the client and API present their certificates to each other for verification and validation. A certificate contains information about the identity of the party and is digitally signed by a trusted certificate party. The communications between the client and the API starts after the successful verification of each party's certificate by the other. Each party validates the received certificate against its trust store. In X509, the message exchanged between the client and API (including the API-Key of the client) are established over secure connections and encrypted using server's private key. This mechanism is mainly used in application-to-application communications and for authorizing access to protected APIs.

DES-TECH Open Authorization Version 2.0 (OAuth 2.0): The objective of OAuth 2.0 authentication is to authorize the access of a client to an API on behalf of an end-user for a limited period. This design is specifically used when the permission and consent of an end-user is required to authorize the access of a client to protected services and resources. For this purpose, when a client application requests for using an API, the API first communicates with the end-user (in a separate communication) to ask for permission. To this objective, either the API provider itself or an authorization server on behalf of the API provider verifies the end-user and ask for her confirmation and permission. Upon the confirmation of the end-user, the client application is allowed to access to the API. To allow temporary access to an API, the API provider issues two types of tokens: (1) Access token and (2) refresh token. To call an API, a client should present these tokens to the API provider. The access token allows the client to access the API once. However, if the authorization server and the end-user allow the client to access the API more than once over a period, a refresh token is also issued for the client application to refresh the access token over the specific period. With a refresh token the client application can access the API without the involvement of the end-user. OAuth 2.0 considers four mechanisms to grant access to an API: (1) Authorization code, (2) Client credentials, (3) End-user credentials, and (4) Implicit. These variants are chosen based on the level of trust in the client application framework.

DES-TECH Open-ID Connect Version 1.0: The objective of OpenID Connect is to authenticate an end entity. The end entity can be an end-user or a client application. For this purpose, an API provider sends a request to an authentication server to verify an end entity. The authentication server communicates with the end-user (in a separate and private connection) and verifies the identity of the end-user. The authentication server informs the API provider of the results of authentication and reveals information about the end-user to the API provider with the consent of the end-user. The API provider receives the information about the end-user in an ID token. Open-ID connect builds on top of OAuth protocol.

DES-OBJ Secure Communications Channels: APIs should use secure communication channels and protocols for receiving requests from the clients and providing response to them. This is specifically used to prevent man-in-the-middle attacks and eaves-dropping attacks.

DES-OBJ Message Encryption: Message encryption mechanisms are responsible for encoding the communications between an API and its clients so that they cannot be read and understood in man-in-the middle attacks or eaves-dropping attacks.

4.5.2 Mechanisms to Address Privacy in APIs

Privacy mechanisms are responsible for preventing end-user's data from being accessed or observed without the permission and consent of the related human parties. Two specific mechanisms that address privacy of APIs include: data masking and end-user notification and approval (Figure 17).

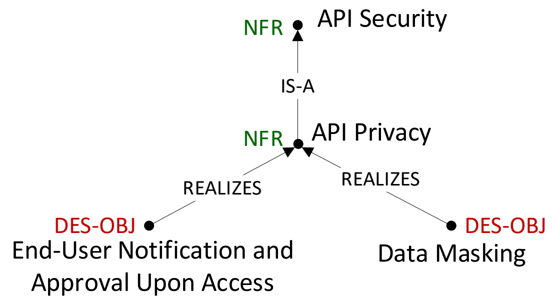


Figure 17 API Privacy Design Techniques

DES-OBJ End-User Notification and Approval: Some of the data that are communicated between an API and the clients may belong to an end-user. In this case, mechanisms need to be designed to inform the end-users that a client is requesting access to their information, to inform the end-users of the possible consequences and usages of their data and to obtain their permission and consent for this kind of data provision.

DES-OBJ Data Masking: Sensitive data and the logs of activities on the confidential data should be hidden or be encrypted by the API provider or by the client before being stored on the back-end systems or on the client side.

4.5.3 Mechanisms to Address Traceability in APIs

Activity logging and user auditing mechanisms keep the detailed record of any communications between an API and a client and help trace the interactions of clients with APIs (Figure 18).

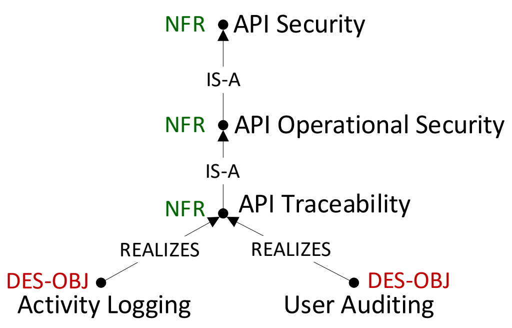


Figure 18 API Traceability Design Techniques

DES-OBJ Activity Logging: Activity logging mechanisms are responsible to log the record of every interaction that occurs between an API and a client. An activity log record includes information about the client, client's type, client's location, the request made by the client, the response provided, and the time and type of requests and response.

DES-OBJ User Auditing: User auditing mechanisms are responsible to provide administrative and historical reports about the users as well as the usage of an API.

4.5.4 Mechanisms to Address Robustness in APIs

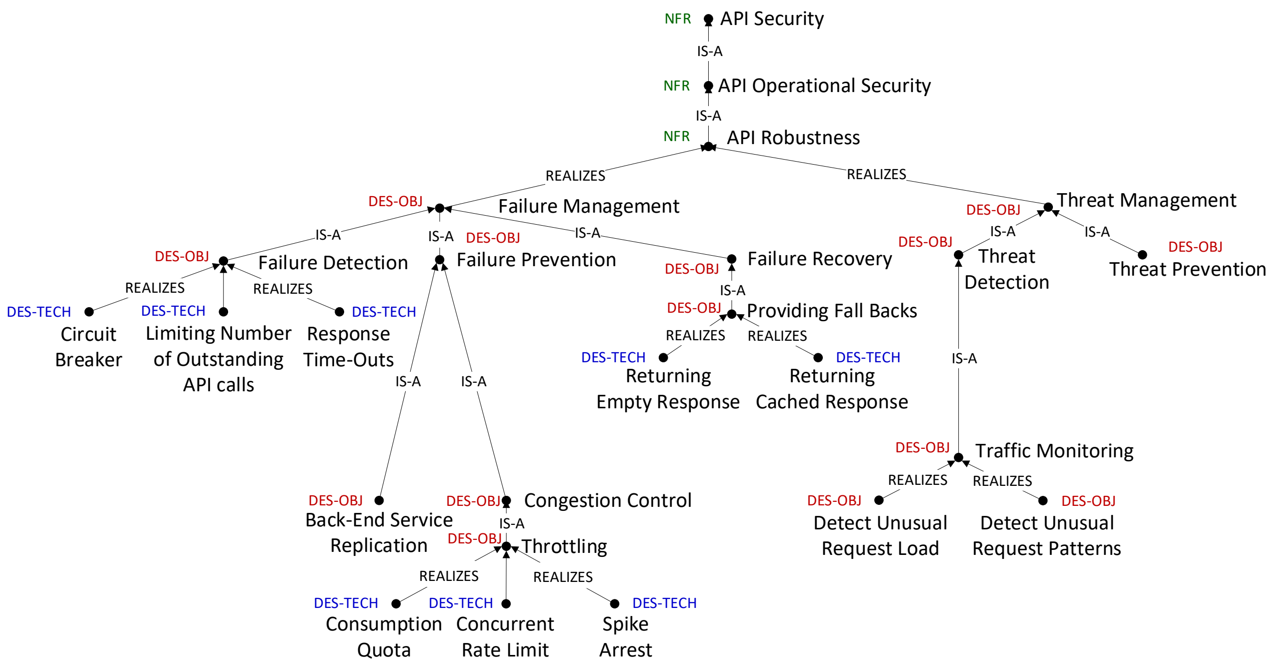
Robustness mechanisms help APIs be resilient against the failure of backend systems and provide continuous service over time. Robustness mechanisms can be divided into two groups: failure management mechanisms and threat management mechanisms (see Figure 19).

DES-OBJ Failure Management: The backend services of an API may become unavailable for a period due to failure, disconnection from the network, or upgrades. Failure management mechanisms are responsible to help APIs manage the failures and unavailability of the back-end systems. Failure management mechanisms can be divided into three group: failure detection, failure prevention, and failure recovery.

DES-OBJ Failure Detection: Failure detection mechanisms are responsible for detecting the failure or unavailability of the backend systems of an API at run-time. Some concrete techniques to design failure detection include response time-outs, circuit breaker pattern, limiting the number of outstanding requests.

DES-TECH Response Timeouts: Response timeouts (also referred to as network timeouts) allow to specify a certain amount of time for responding to API requests. Defining timeouts for an API allows the clients to detect failure in the operation of the API and its backend services and not to infinitely wait for a response.

DES-TECH Circuit Breaker: In circuit breaker pattern, the number of successful and failed responses are recorded. If the rate of failed responses exceeds a pre-defined threshold, the circuit will become open and further requests fail immediately since the threshold shows that it is highly possible that the back-end services are not available. After the timeout, the clients try again and when the response is provided successfully, the circuit will be closed again.



NFR API Security
IS-A
NFR API Operational Security
IS-A
NFR API Robustness
REALIZES
REALIZES
DES-OBJ Failure Management
DES-OBJ Threat Management
IS-A
IS-A
IS-A
DES-OBJ Failure Detection
DES-OBJ Failure Prevention
DES-OBJ Failure Recovery
DES-OBJ Threat Detection
DES-OBJ Threat Prevention
REALIZES
REALIZES
DES-TECH Circuit Breaker
DES-TECH Limiting Number of Outstanding API calls
DES-TECH Response Time-Outs
IS-A
DES-OBJ Providing Fall Backs
REALIZES
REALIZES
DES-TECH Returning Empty Response
DES-TECH Returning Cached Response
IS-A
IS-A
IS-A
DES-OBJ Traffic Monitoring
REALIZES
REALIZES
DES-OBJ Detect Unusual Request Load
DES-OBJ Detect Unusual Request Patterns
DES-OBJ Back-End Service Replication
DES-OBJ Congestion Control
IS-A
DES-OBJ Throttling
REALIZES
REALIZES
DES-TECH Consumption Quota
DES-TECH Concurrent Rate Limit
DES-TECH Spike Arrest

Figure 19 API Robustness Design Techniques

DES-TECH Limiting the Number of Outstanding Requests: Limiting the number of clients' requests imposes an upper bound on the maximum number of waiting requests that clients can have for a particular service. All subsequent requests are failed immediately if the limit is exceeded since it is highly likely that the API and its backend services are unavailable.

DES-OBJ Failure Prevention: Backend systems of an API fail momentarily or permanently over time due to various reasons, such as disconnection from the network, upgrades, or corruption. Failure prevention mechanisms are often designed in the architecture of web APIs to help service continuity of APIs in the face of failure. Two specific types of failure management mechanisms include: congestion control mechanisms and back-end service replication.

DES-OBJ Congestion Control: API congestion control mechanisms are responsible for monitoring and controlling the traffic of the calls that are made to an API. These mechanisms proactively monitor and control the number of clients' requests to an API and if the traffic exceeds the capacity of the API and the backend services, they return the traffic to a level that can be handled by the API. API throttling mechanisms are specifically designed to slow down and keep the traffic of requests at a manageable level. Some concrete techniques to throttle the traffic of requests include spike arrest policy, consumption quota, and concurrent rate limit.

DES-TECH Spike Arrest Policy: The objective of spike arrest policy is to limit the total amount of calls that can be made to an API by all the clients over a period of time. To this objective, spike arrest distributes the traffic of API calls evenly among equal time frames. If the traffic of API calls is above the limit of spike arrest in a time frame, then the request will be dropped and not be responded by the API. This mechanism is mainly suggested to control the traffic of API calls during peak hours and to protect the API from unexpected bursts in the requests.

DES-TECH Consumption Quota: The objective of consumption quota (also referred to as rate limit) is to keep the traffic of API calls that are made by each client application within the capacity of the API. For this purpose, a quota is defined for each client limiting the numbers of calls that the client can make over a specific period. If the number of the calls made by client exceeds the limit, they will be dropped off and not responded. Quota limits the requests of each client but has no control over the total amount of requests made by all the clients.

DES-TECH Concurrent Rate Limit: The objective of concurrent rate policy mechanism is to limit the number of simultaneous connections that can be made by an API to the back-end systems. If the number of connections exceeds the limit then the clients' requests are rejected.

DES-OBJ Backend Service Replication: Replication and redundancy is a common group of mechanisms to prevent the failure. In these mechanisms, the backend services and systems are duplicated and sometimes distributed geographically so that once one instance of the service fails, other instances can provide service.

DES-OBJ Failure Recovery: The backend systems of API may fail momentarily and permanently. Mechanisms need to be in place that allow the API to manage requests until the backend systems return to service. One mechanism to handle recovery period is to provide fallbacks.

DES-OBJ Providing Fallbacks: Alternative and back-up plans should be in place when the API provider or the back-end services fail and cannot respond to the clients. Two concrete techniques to provide alternative plans include: returning empty responses to the clients during failed time and returning cached responses.

DES-TECH Returning Empty Responses: In case of failure, the API returns responses with empty body.

DES-TECH Returning Cached Responses: In case of failure, the API returns cached responses to similar requests.

DES-OBJ Threat Management: Malicious clients may attack an API in various ways and damage the health and operation of the API and the backend systems. Threat management mechanisms need to be in place that protect an API against potential attacks of malicious clients.

DES-OBJ Threat Detection: Threat detection mechanisms are responsible for monitoring and proactively detecting any kind of malicious access to an API and the back-end systems. Two threat protection mechanisms include: detecting unusual request patterns and detecting unusual requests loads for an API.

DES-OBJ Detecting Unusual Request Patterns: The goal of these mechanisms is to detect unusual access patterns to an API, and to block these requests or provide alert for them.

DES-OBJ Detecting Unusual Request Loads: The goal of these mechanisms is to detect unusually high number of API calls in the API requests, and to block these requests or generate alert for them.

DES-OBJ Threat Prevention: Threat prevention mechanisms are responsible for monitoring and detecting any kind of malicious access to an API and the back-end systems.

5 Trade-Offs of the Design Techniques

We extracted and structured the trade-offs of 22 design techniques into the body of WEBAPIK as reviewed in the following.

5.1 Trade-Offs of API Access Authorization Techniques

The trade-offs of access authorization techniques are summarized in Figure 20 to Figure 24 and described in the following.

5.1.1 API-Key Trade-Offs

EFFECT Accessibility – Access Simplicity: (+) (Strong). The potential clients can easily access the API and the back-end system by registering with the API provider and obtaining an API key. The only barrier and security check that is performed to access an API is the providing the key which can be easily obtained.

EFFECT Usability – Usage Simplicity: (+) (Strong). An API can be simply used by presenting a key to the API provider. There are low security barriers to use an API.

EFFECT Performance – Latency: (+) (Strong). Low overhead is added to the communications between a client and an API provider to allow access to an API.

EFFECT Security – Access Confidentiality: (+) (Weak). API-Key provides low level of confidentiality for accessing to an API. The client (the provider of the key) is only authorized and not authenticated. Since API-Key is not encrypted or signed, it is exposed to eaves-dropping attacks.

EFFECT Security – Message Confidentiality: (-) (Strong). The API-key is usually not encrypted. This exposes the key to eaves dropping attacks.

EFFECT Privacy: (-) (Strong). The API-key does not have any mechanism to ensure the privacy of the related human users upon access to the API.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [11], [76], [77].

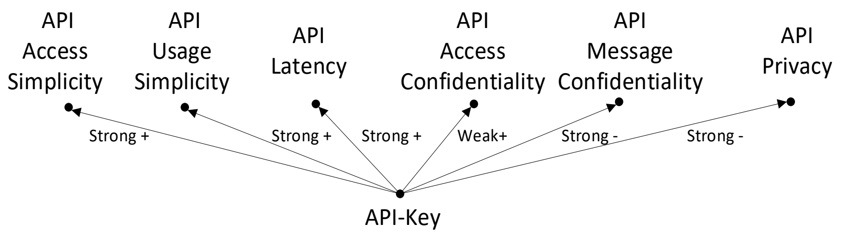


Figure 20 API-Key Trade-Offs

5.1.2 Username and Password Trade-Offs

EFFECT Accessibility – Access Simplicity: (+) (To Some Extent): The end user and the client application can easily access the API via presenting the end-user's credentials.

EFFECT Usability – Usage Simplicity. (+) (Strong): The only barrier and security overhead to use an API is to provide the end-user's credentials. Username and password provide a basic and simple mechanism to access an API.

EFFECT Performance – Latency: (-) (To Some Extent): Low overhead is added to the communications between the client and the API provider to authenticate the end user. Only one round of additional interactions between the client and the API provider is required to access an API in which a human user is also involved. However, encryption and decryption of the end-user's credential adds additional operations and time to these interactions.

EFFECT Security – Access Confidentiality: (+) (Weak): Username and password is the minimum mechanism that can be used to verify the identity of the end user and authorize access.

EFFECT Security – Message Confidentiality. (+) (To Some Extent). Encrypting the credentials has positive impact on the confidentiality of the messages that are communicated between a client and an API provider.

EFFECT Privacy. (+) (Weak): Obtaining user's credential helps obtain the permission and consent of the end user before giving access to the data, information and services that belong to the user.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [11].

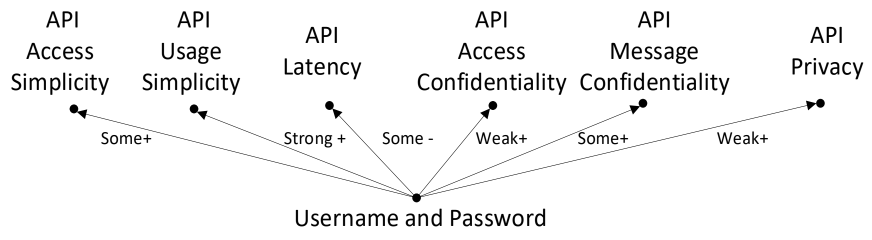


Figure 21 Username and Password Trade-Offs

5.1.3 Certificate-Based Authentication Mechanism X.509 Trade-Offs

Accessibility – Access Simplicity: (-) (Strong): Two-way and mutual authentication creates difficult barriers to access and API and is specifically used for obtaining access to protected APIs. Both the client and the API provider should authenticate and verify the certificate of each other before they can communicate. Moreover, obtaining a certificate to access the API is not an easy process for clients and requires interactions with identity providers.

EFFECT Usability – Usage Simplicity: (-) (Strong): The client applications and services require to obtain a certificate to be able to use the API. Using public and private keys and obtaining certificates by both the client and the API provider makes the API difficult to use.

EFFECT Performance – Latency: (-) (To Some Extent): The additional rounds of interactions between the client and the API provider required to validate each other and the encryption and decryption of the message exchanges have negative impacts on the response time of the API, and specifically on the time that is required to set up an access to an API. Additional rounds of interactions between a client and an API provider have stronger negative impact than the time required to encrypt and decrypt the communicated messages.

EFFECT Security – Access Confidentiality: (+) (Strong): Two-way and certificate-based authentication increases the security of access to an API and increases the trust of each party in the authenticity of the other. This technique is mainly used to provide access to protected APIs.

EFFECT Security – Message Confidentiality: (+) (Strong): Using secure connections between the client and the API providers and encrypting the messages using public and private keys helps protect the confidentiality of the interactions between the client and the API provider.

EFFECT Security – Privacy. (-) (Strong): This technique does not consider any mechanism to obtain the consent of the data or service owner before authorizing access to the API.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [11].

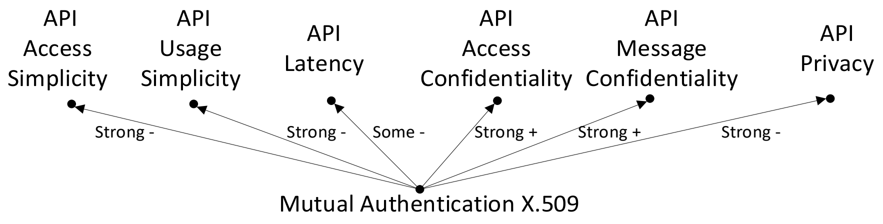


Figure 22 Mutual *Authentication X.509* Trade-Offs

5.1.4 Open Authorization Version 2 Trade-Offs

EFFECT Accessibility – Access Simplicity. (-) (To Some Extent): OAuth 2.0 involves end-user in permitting access to an API. Even if the end-user permits, the access to the API will be temporary and over a specific period. In OAuth 2.0, the level of access can be controlled using the refresh tokens which identify the allowed period of access and the number of accesses to an API. When the refresh token expires, the client should re-obtain the permission of the end-user.

EFFECT Usability – Usage Simplicity. (-) (To Some Extent): In OAuth 2.0, several rounds of interactions are required between the client and the API provider to obtain access or refresh token to call an API. These interactions should be repeated once the token expire after a specific period.

EFFECT Performance – Latency. (-) (To Some Extent): Involvement of the end-user (a human user) in the authorization process significantly impacts the waiting time of a client application to obtain access and call an API.

EFFECT Security – Access Confidentiality: (+) (To Some Extent). Access to an API is authorized with the permission of the resource owner. Moreover, the resource owner who permits access to the API is authenticated via presenting credentials. The client application will receive encrypted tokens upon the permission. These checks have positive impact on access confidentiality to some extent.

EFFECT Security – Message Confidentiality: (+) (To Some Extent): Interacting over secure connections as required in O-Auth 2.0 helps protect the confidentiality of communications between the client and the API provider to some extent.

EFFECT Security – Privacy: (+) (To Some Extent): The API provider obtains the permission and consent of the end-user before allowing a client application to access an API. Moreover, the access permission is temporary and is valid for a specific period. Therefore, the end-user can decide when to stop giving access to a client application. However, during the period that the refresh tokens are valid, clients are allowed to access the API without asking the user for a period for more than once. This has some negative effect on the privacy of users.

Type of Supporting Evidence: Qualitative reasoning, expert opinion[10], [11], Formal Proof [83], and Empirical Evaluation[17], [84], [85], [86], [87], [88].

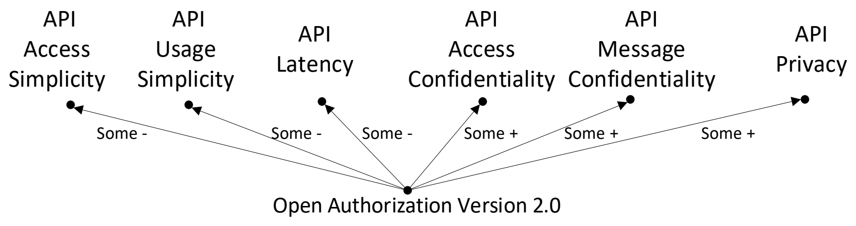


Figure 23 Open Authorization Version 2.0 Trade-Offs

5.1.5 OpenID Connect Trade-Offs

EFFECT Accessibility – Access Simplicity: (-) (To Some Extent): Involvement of the end-user and authentication server in the authorization and authentication process creates barrier for accessing an API.

EFFECT Usability – Usage Simplicity: (-) (To Some Extent)**:** In OpenID Connect, a client should interact with both an end-user and an authentication server. These interactions harden the use of an API.

EFFECT Performance – latency: (-) (Strong): Additional rounds of interactions between an end-user and an authentication server are required to verify the identity of the end-user. The involvement of the end-user and an authentication server in the authentication and authorization process significantly impacts the latency of an API.

EFFECT Security – Access Confidentiality: (+) (To Some Extent): Verifying the identity of an end user increases the confidence of the API provider in the end-user who is permitting access to an API. Moreover, since the identity of the end-user is confirmed by a party that is trusted by both the client and the API provider, both parties can have confidence in the authentication results to some extent.

EFFECT Security – Message Confidentiality: (+) (To Some Extent): Communications between clients and API provider and occur over secure connection channels. Secure channels help ensure the confidentiality of the exchanged messages to some extent.

EFFECT Security – Privacy: (+) (To Some Extent): The permission and consent of the end-user is obtained to expose end-user's information to a client application. However, the end-user may have little control over what identity information is revealed to the authentication server, or to the API provider. Moreover, when the user has several interactions with an authentication server, the authentication server may obtain unauthorized knowledge about the interactions of the end user.

Type of Supporting Evidence: Qualitative reasoning, expert opinion [82], Formal Proof [93], and Empirical Evaluation [18], [89], [90], [91], [92], [94], [95], [96].

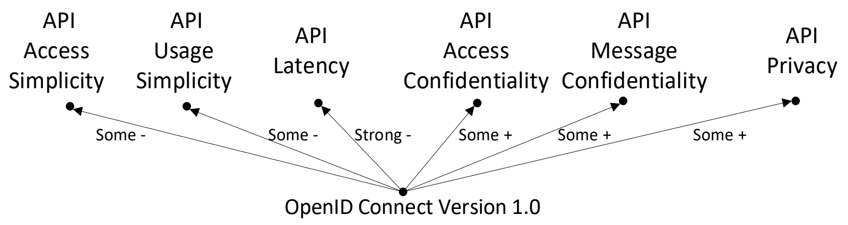


Figure 24 OpenID Connect Version 1.0 Trade-Offs

5.2 Trade-Offs of API Throttling Techniques

The trade-offs of API Throttling techniques are summarized in Figure 25 to Figure 27 and described in the following.

5.2.1 Spike Arrest Policy Trade-Offs

EFFECT Performance – Availability: (-) (To some Extent): In spike arrest policy, there are times that the API does not respond to the clients' requests and the requests are dropped off. This negatively affect the availability of the API.

EFFECT Performance – Latency: (-) (To Some Extent): The API may not respond in a timely manner to the clients if the number of API calls is above the threshold. Client needs to call the API until it becomes available.

EFFECT Security – Robustness: (+) (Strong): Controlling the traffic of API calls and keeping the traffic limit within the capacity of API helps control the request load of the API and the backend services and ensures healthy operation of backend services. Moreover, controlling the traffic of API calls helps the continuous operation and stability of the API and back-end systems over time. Smoothing out the traffic helps protect the API against denial-of-service attacks.

EFFECT Security – Reliability – Integrity (-) (To Some Extent): It is not guaranteed that the clients' requests will be responded by the API. The calls made beyond the limit of spike arrest will be dropped off. It is not ensured that the client will receive a response from an API.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [81]

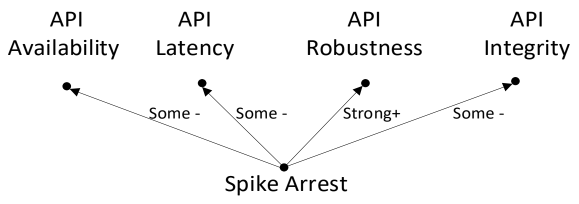


Figure 25 Spike Arrest Trade-Offs

5.2.2 Consumption Quota Trade-Offs

EFFECT Accessibility – Access Frequency (-) (To Some Extent): The access of the clients to an API is limited to their quota. The clients do not have access to the API when the number of calls exceeds the client's quota.

EFFECT Security – Robustness. (+) (To Some Extent): Defining a consumption quota for each client helps keep the requests of client within the capacity of the API and the back-end systems and helps avoid denial of service-attacks. However, it does not control the bursts in requests traffic.

EFFECT Security – Reliability – Integrity. (+) (Strong): Since a client knows how many calls it can make to an API; it is assured that the calls made within the specified quota will be responded by the API provider.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [77], [81].

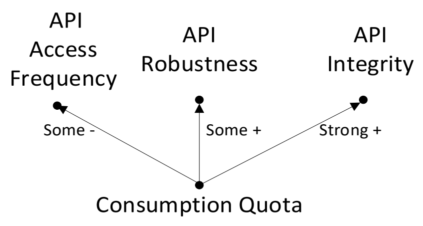


Figure 26 Consumption Quota Trade-Offs

5.2.3 Concurrent Rate Limit Trade-Offs

EFFECT Performance– Availability: (-) (To Some Extent): Rate control affects the availability of API services. If the number of simultaneous requests exceeds the concurrency rate, the requests of clients will be dropped off.

EFFECT Performance – Latency: (-) (To Some Extent): It is not guaranteed that the API responds the requests in a timely manner. If the requests of clients exceed the concurrency rate, they will not be responded. Hence, the clients should attempt several times to call the API until they receive a response.

EFFECT Security – Robustness: (+) (Strong): Controlling the number of connections to the back-end systems positively contributes to the service continuity and stability of the API and back-end services and ensures that the request load will not exceed the capacity of the backend systems.

EFFECT Security – Reliability – Integrity: (-) (To Some Extent): It is not guaranteed that the requests of clients will be responded by the API. Those calls that pass the concurrent rate limit will be dropped off.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [81].

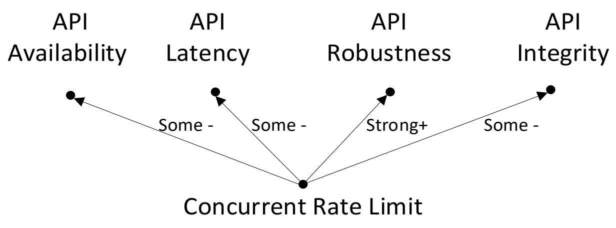


Figure 27 Concurrent Rate Limit Trade-Offs

5.3 Trade-Offs of API Communication Techniques

The trade-offs of API communication techniques are summarized in Figure 28 to Figure 31 and described in the following.

5.3.1 Trade-Offs of One-to-One Synchronous Communication Mechanism

EFFECT Performance - Throughput: (-) (Strong): The back-end services process the requests of the clients one by one and in the order of the arrival of the requests. Responding the incoming requests begins when the current requests are responded. This one-by-one and in-order processing significantly impacts the throughput of the API.

EFFECT Performance – Latency: (+) (Strong): The clients are assured and guaranteed that they will receive a response within a predefined time limit.

EFFECT Security – Robustness (-) (To Some Extent): In synchronous communications, there is no failure prevention mechanism. If the API is not available for some time, the requests will be dropped off and no response will not be provided.

EFFECT Security – Reliability – Integrity: (+) (To Some Extent): The clients know how long to wait to receive a response. If a response is not received within the pre-determined time limit, the clients can follow up with the request.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [39].

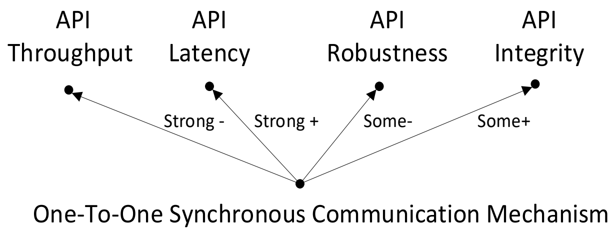


Figure 28 One-To-One Synchronous Communication Trade-Offs

5.3.2 Trade-Offs of One-to-One Asynchronous Communication Mechanism

EFFECT Performance – Throughput: (+) (Strong): The request and response queues in the asynchronous communication decouple the behavior and operation of clients from the API provider. Since the API provider can decide when to respond to the clients' request, the requests does not interrupt the operation of the back-end systems. Moreover, the clients' requests do not need to be responded one by one in the order of arrival.

EFFECT Performance – Latency: (-) (To Some Extent): Since the requests are not responded in the order of arrival, there is no guarantee that the clients will receive a response in a reasonable or prespecified time limit. It is also possible that the requests wait in the request queues for unlimited amount of time or be dropped of the request queues when the queues are full.

EFFECT Security - Robustness (+) (To Some Extent): The request queue acts as a buffer between the client and the back-end services. If the back-end services are not available for some time, the requests can be queued up and be responded later after the recovery of the back-end systems.

EFFECT Security – Reliability - (Integrity) (-) (To Some Extent): If the request queue gets full (i.e., the size of request queue is smaller than the number of requests that arrive in a time unit), the subsequent requests will be dropped off the queue and not responded.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [39].

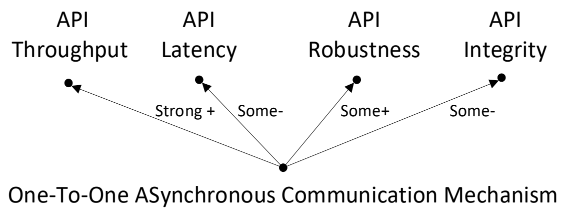


Figure 29 One-To-One Asynchronous Communication Trade-Offs

5.3.3 Trade-Offs of (One-to-One) Synchronous to Asynchronous Communication Mechanism

EFFECT Performance – Throughput (+) (Strong): Since only the interface of the API provider is synchronous and the order in which the clients' requests will be responded can be controlled by the API provider, the throughput of the API can be controlled.

EFFECT Performance - Latency (+) (To Some Extent): In synchronous to asynchronous mechanism, the API provider guarantees to provide a response within a specified time limit while the back-end services work asynchronously. This has positive impact on the latency of the API.

EFFECT Security –Robustness (+) (Strong): The API acts as a buffer between the client and the back-end services. If the back-end services are not available for some time, the requests can be queued up and be responded later although the client is communicating synchronously.

EFFECT Security – Reliability – Integrity. (+) (Strong): In this mechanism, there is a guarantee for delivery of the clients' requests to the back-end services and for responding the clients' request. The API will try until it receives the response of the backend services and sends the response back to the clients. If the back-end services are not available for some time the clients' requests will be queued.

EFFECT Interoperability – Flexibility in Communication Protocol (+) (To Some Extent): This mechanism allows clients with different communication mechanisms communicate with the back-end services. The communication style of the clients can be different from the back-end services.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [65].

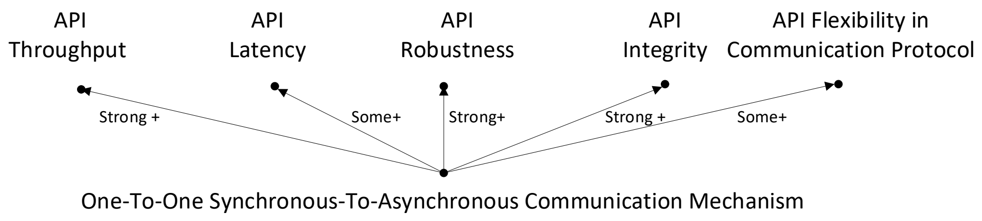

API Throughput
API Latency
API Robustness
API Integrity
API Flexibility in Communication Protocol
Strong +
Some+
Strong+
Strong +
Some+
One-To-One Synchronous-To-Asynchronous Communication Mechanism

Figure 30 One-To-One Synchronous-To-Asynchronous Communication Trade-Offs

5.3.4 Trade-Offs of One-to-Many Publish and Subscribe Mechanism

EFFECT Accessibility – Access Simplicity: (+) (To Some Extent): Subscribers can access the messages by registering and subscribing to the publisher. Subscribing to the publisher is done once upon registration, and the subscribers can access the related messages easily.

EFFECT Extensibility – Server-Side Extensibility and Client-Side Extensibility: (+) (Strong): In this mechanism, new back-end services and clients can be easily added or removed from the system at run-time. Clients and back-end services have little or no information about each other or the topology of the system or when an update happens. Thus, publishers and subscribers are decoupled both in terms of location and time.

EFFECT Performance – Response Time: (-) (To Some Extent): Publishers become an intermediate between the clients and the back-end services, and additional interactions should happen to inform the target audience of the updates. These additional interactions may take up some additional time.

EFFECT Security – Access Confidentiality: (+) (Weak): The access to the data can be controlled centrally by the publisher.

EFFECT Security – Reliability - Integrity: (-) (To some extent): It is not guaranteed that the published messages are received by all the subscribers.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [39], and empirical evaluation [97], [98], [100], [101], [102], [103]

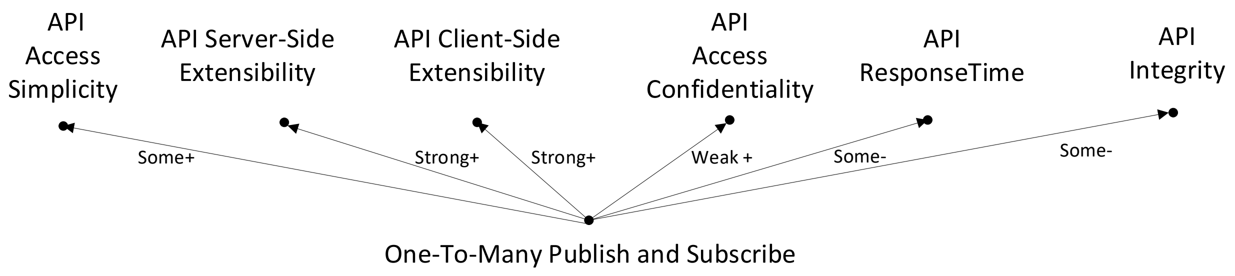

API Access Simplicity
API Server-Side Extensibility
API Client-Side Extensibility
API Access Confidentiality
API ResponseTime
API Integrity
Some+
Strong+
Strong+
Weak +
Some-
Some-
One-To-Many Publish and Subscribe

Figure 31 One-To-Many Publish and Subscribe Trade-Offs

5.4 Trade-Offs of API Gateway Design Techniques

The trade-offs of API gateway design techniques are summarized in Figure 32 to Figure 33 and described in the following.

5.4.1 Central API Gateway Trade-Offs

EFFECT Server-Side Extensibility (+): (Strong): API gateway hides the details of the back-end services, how they work and how they provide service. Thus, the back-end services can change easily without affecting the clients.

EFFECT Performance – Throughput (-): (To Some Extent): A central gateway becomes the single point of access to the back-end systems and becomes a bottleneck specifically when the load of clients' requests is high.

EFFECT Performance – Response Time (-): (To Some Extent): A central gateway adds another component (an intermediary) between the clients and the back-end services, increasing the number of interactions required to respond to a single request.

EFFECT Security – Access Confidentiality (+): (To Some Extent**)**: Since clients interact with the API gateway, The API gateway protects the back-end systems and services from direct access and hides the information and the details of back-end systems from the clients.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [14], [103].

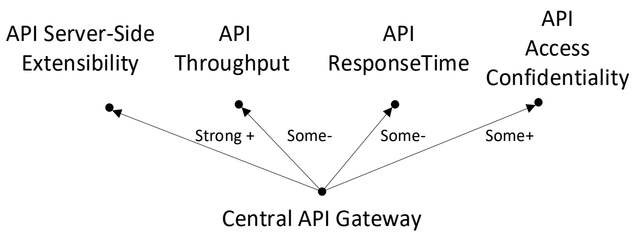


Figure 32 Central Gateway Trade-Offs

5.4.2 Multiple API Gateways (Back-End for Front-End) Trade-Offs

EFFECT Server-Side Extensibility (+): (Strong): API gateways hide all the details of the back-end services, how they work and how they provide service. The back-end services can change easily without affecting the clients.

EFFECT Performance – Throughput (+): (To Some Extent): The load of requests is distributed between multiple gateways each of which address a separate group of clients and improves the throughput of the API.

EFFECT Performance – Response Time (-): (To Some Extent): Gateways adds another component (an intermediary) between the clients and the back-end services, increasing the number of interactions required to respond to a single request.

EFFECT Security – Access Confidentiality (+): (To Some Extent): Since the clients interact with the API gateways, the gateways protect the back-end systems and services from direct access.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [14], [59]

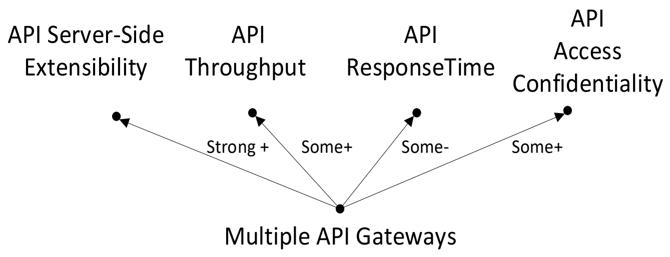


Figure 33 Multiple API Gateways Trade-Offs

5.5 Trade-Offs of Service-Registration Techniques

The trade-offs of service registration techniques are summarized in Figure 35 to Figure 34 and described in the following.

5.5.1 Self Registration Trade-Offs

EFFECT Usability – Usage Simplicity: (-) (To Some Extent): In self registration, all the burden and responsibility of service registration is shifted towards the clients (the services that want to register themselves as service provider).

EFFECT Server-Side Extensibility (+): (To Some Extent): Clients can easily register and deregister themselves as service provider at run-time. The run-time registration has positive impact on the extensibility of the service-providers.

EFFECT Security – Access Confidentiality (-): (To Some Extent): The details of service registration and the service registry is exposed to the clients. Clients have direct access to the service registry and have visibility into the details of service registration. This direct access and visibility may be misused by malicious clients.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [62], [63].

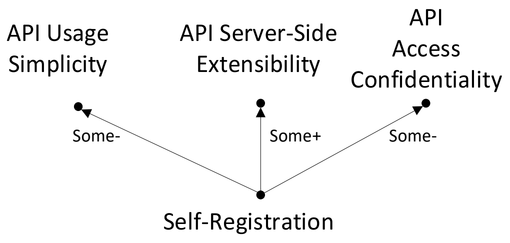


Figure 34 Self-Registration Trade-Offs

5.5.2 (Server-Side) Third-Party Registration Trade-Offs

EFFECT Usability – Usage Simplicity: (+) (Strong): Since the third-party registrar or the API provider are responsible for registering, looking up, and deregistering the back-end services, all the burden of registration is removed away from the clients (i.e., the services that want to register themselves).

EFFECT Server-Side Extensibility (+): (Strong): Server-side or third-party registrations hides the details of service registration and the service registry table from the clients. The registration mechanism can easily change without affecting the clients.

EFFECT Security – Access Confidentiality (+): (To Some Extent): The details of service registration and the service registry is controlled by the API provider. The clients do not have any information and direct access to the service registry. This protects the API provider from malicious clients.

Type of Supporting Evidence: Qualitative reasoning and expert opinion [62], [64].

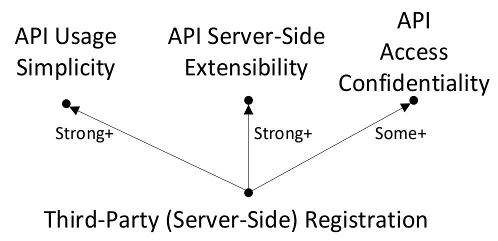

API Usage Simplicity
API Server-Side Extensibility
API Access Confidentiality
Strong+
Strong+
Some+
Third-Party (Server-Side) Registration

Figure 35 Third-Party (Server-Side) registration Trade-Offs

5.6 Trade-Offs of Service Discovery Techniques

The trade-offs of access service discovery techniques are summarized in Figure 34 to Figure 37 and described in the following.

5.6.1 Trade-Offs of Server-Side Service Discovery

EFFECT Usability – Usage Simplicity: (+) (Strong): In server-side discovery and service mapping, the burden of finding instances of the related services is shifted towards the API. Thus, a single and easier-to-use point of access to a complicated backend system is exposed towards the clients.

EFFECT Server-Side Extensibility (+): (Strong): Server-side discovery hides the details of the back-end services, their location, as well as discovery and load balancing mechanisms from the clients. Thus, backend services can easily be added, removed, or modified without affecting the clients.

EFFECT Performance – Latency (+): (Strong): Server-side discovery reduces the number of interactions between the client and the server. Minimizing the interactions between the clients and the API provider, has strong positive effect on the latency of web APIs.

EFFECT Performance – Throughput: (+) (To Some Extent): Server-side service discovery help reduce the number of interactions between a client and an API, reduce the request load of the API provider, and finally increase the number of requests that can be responded per time unit.

EFFECT Security – Access Confidentiality: (+) (To Some Extent): The internal logic of the back-end services remains hidden from the clients. Hiding the logic of operation protects the back-end services from security attacks.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [62], [68], [103], [104].

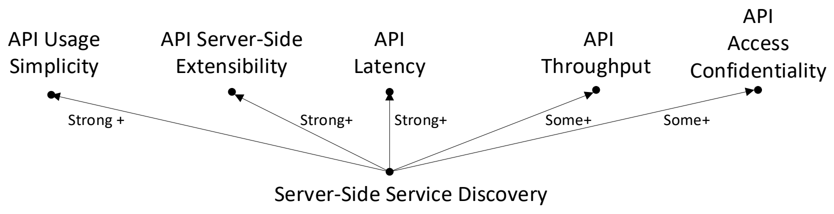

API Usage Simplicity
API Server-Side Extensibility
API Latency
API Throughput
API Access Confidentiality
Strong +
Strong+
Strong+
Some+
Some+
Server-Side Service Discovery

Figure 36 Server-Side Service Discovery Trade-Offs

5.6.2 Trade-Offs of Client-Side Service Discovery

EFFECT Usability – Usage Simplicity: (-) (To Some Extent): All the burden and responsibility of service discovery, load balancing and routing is shifted towards the clients. Clients should interact with several finer grained APIs or should implement some part of load balancing logic themselves to find the right instance of the back-end services. This makes the use of an API difficult.

EFFECT Server-Side Extensibility (+): (To Some Extent): Service discovery helps service instances dynamically change and be added or removed at run-time. This has positive impact on server-side extensibility.

EFFECT Performance – Latency (-): (To Some Extent): In client-side service discovery, multiple rounds of interaction should be performed to find an appropriate service instance. This imposes additional delays before a client can access a service.

EFFECT Performance – Throughput: (-) (To Some Extent): Multiple rounds of interactions between clients and the API provider is required to an answer a client's request. These additional interactions which can be slow over the web has negative impact on the throughput of the API provider.

EFFECT Security – Access Confidentiality: (-) (To Some Extent): The internal details of the back-end services is visible and exposed to the clients. Exposing the logic of operation towards clients increases the chances of malicious misuse.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [62], [69], [103], [104].

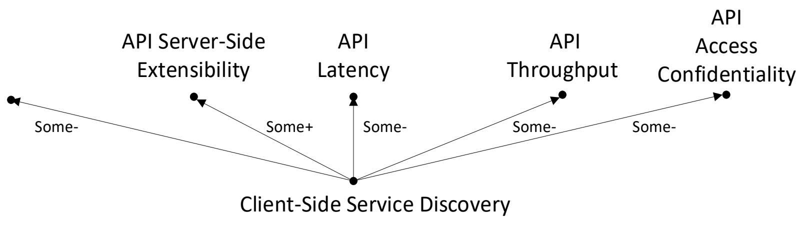

API Server-Side Extensibility
API Latency
API Throughput
API Access Confidentiality
Some-
Some+
Some-
Some-
Some-
Client-Side Service Discovery

Figure 37 Client-Side Service Discovery Trade-Offs

5.7 Trade-Offs of API Mapping and Composition Techniques

The trade-offs of API mapping and composition techniques are summarized in Figure 38 to Figure 39 and described in the following.

5.7.1 Trade-Offs of Server-Side API Mapping and Composition

EFFECT Usability – Usage Simplicity (+) (Strong): Server-Side API composition increases the granularity of the provided services to the clients and allows to expose a simple and single API that hides the complexity of the involved back-end services.

EFFECT Server-Side Extensibility: (+) (Strong): API composition encapsulates the details of the back-end system from the client and minimizes the interactions between the client and the backend services. The backend services can be easily extended, added, and removed without involving the clients.

EFFECT Performance – Latency: (+) (Strong): API composition helps reduce the number of interactions between a client and an API to respond to a single request. Reducing the number of interactions is important in web APIs since interactions over internet may have considerable delays.

EFFECT Performance – Throughput: (+) (To Some Extent): API composition helps reduce the number of interactions between a client and an API, reduces the request load of the API provider and thus increases the number of requests that can be answered per time unit.

EFFECT Security – Access Confidentiality: (+) (To Some Extent): The details of the backend services remain hidden from the clients. Hiding the logic of operation protects the back-end services from security attacks.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [65], [66], [67].

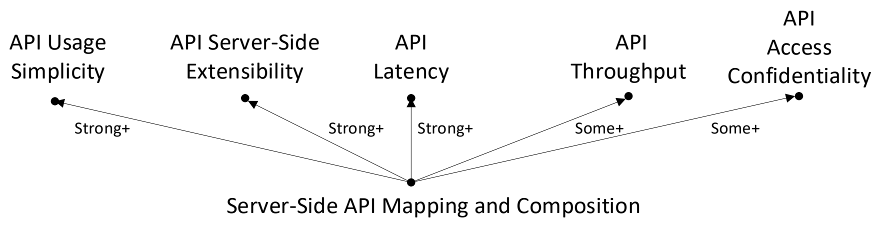

API Usage Simplicity
API Server-Side Extensibility
API Latency
API Throughput
API Access Confidentiality
Strong+
Strong+
Strong+
Some+
Some+
Server-Side API Mapping and Composition

Figure 38 Server-Side API Mapping Trade-Offs

5.7.2 Trade-Offs of Client-Side API Mapping and Composition

EFFECT Usability – Usage Simplicity (-) (To Some Extent): In this design, the burden of API composition is shifted towards the clients. The clients should understand the logic of several fine-grained APIs, find and interact with the related APIs to receive a response to their requests. Shifting composition responsibility makes the use of the API difficult.

EFFECT Server-Side Extensibility: (+) (To Some Extent): The backend services and APIs can change dynamically over time and be added or removed easily at run-time.

EFFECT Performance – Latency: (-) (To Some Extent): To receive a response, clients should interact in several rounds with the related API and make several calls. These interactions increase the latency of receiving a response.

EFFECT Performance – Throughput: (-) (To Some Extent): Multiple rounds of interactions between clients and the API provider is required to an answer a single request. These additional interactions which may be slow over the web has negative impact on the throughput of the API provider.

EFFECT Security – Access Confidentiality: (-) (To Some Extent): The back-end APIs are exposed towards the client. This exposure may create potential security threats since back-end APIs are visible towards the clients.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [65], [66], [67].

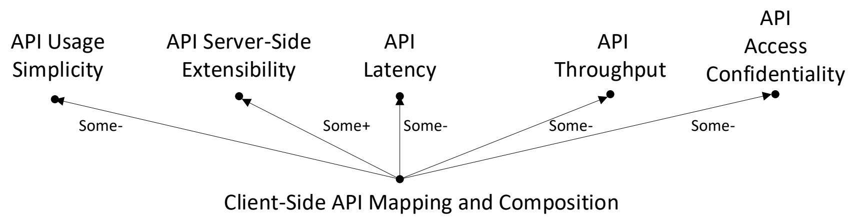

API Usage Simplicity
API Server-Side Extensibility
API Latency
API Throughput
API Access Confidentiality
Some-
Some+
Some-
Some-
Some-
Client-Side API Mapping and Composition

Figure 39 Client-Side API Mapping Trade-Offs

5.8 Trade-Offs of Service Orchestration Design Techniques

The trade-offs of service orchestration techniques are summarized in Figure 40 to Figure 41 and described in the following.

5.8.1 Trade-Offs of Server-Side Two-Phase Transaction Management Technique

EFFECT Usability – Usage Simplicity (+) (Strong): Hiding the complexity of the back-end systems facilitates the use of API by reducing the responsibility of the client in using an API.

EFFECT Server-Side Extensibility: (+) (To Some Extent): Server-Side coordination reduces the coupling between the client and the backend systems. Backend services can dynamically change at runtime without involving the clients.

EFFECT Performance – Latency (+) (Strong): Coordination of the backend services by the API provider reduces the number of interactions between the client and the API provider, reduces the number of API calls required to receive a response, and has positive effect on the latency of an API.

EFFECT Performance – Throughput (+) (Strong): Coordination of the backend services reduces the number of interactions between the client and the API provider and reduces the number of API calls. Reducing the number of API calls reduces the request load of the API provider.

EFFECT Security – Robustness (+) (To Some Extent): It is possible that some of the backend services are not available for a while. As a result, the coordination should resume and continue until all the involved services are available. Coordinating the services on the API provider side avoids these problems to be visible to the client.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [65]

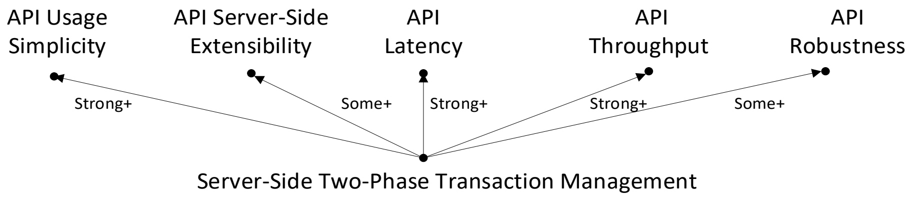


Figure 40 Server-Side Two-Phase Transaction Management Trade-Offs

5.8.2 Trade-Offs of Client-Side Two-Phase Transaction Management Technique

EFFECT Usability - Usage Simplicity (-) (Strong): Shifting the responsibility of synchronizing and orchestrating the related services towards clients strongly impacts the usability of the provided service. If any of the related back-end services fail, the client should repeat multiple rounds of interactions to finally receive a response.

EFFECT Server-Side Extensibility: (+) (To Some Extent): Instances of backend services can change dynamically at run-time.

EFFECT Performance - Latency (-) (Strong): Additional interaction and orchestration load is imposed to receive a final response. This interaction load significantly lengthens the time to receive a response.

EFFECT Performance – Throughput (-) (Strong): Multiple additional rounds of interactions between the API provider and the clients are required to synchronize the related backend services. If any of the related services is unavailable, all the interactions to orchestrate should be repeated. This significantly impact the number of requests that can be responded per time unit.

Type of Supporting Evidence: Qualitative reasoning, and expert opinion [65].

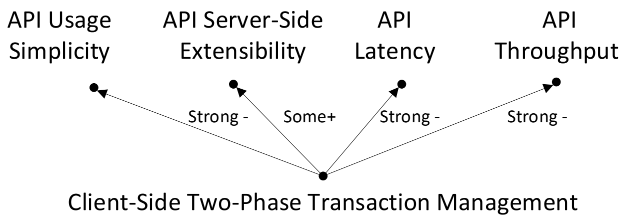


Figure 41 Client-Side Two-Phase Transaction Management Trade-Offs

6 Discussion

In this study, we have extracted, collected, and structured a body of design knowledge about addressing non-functional requirements in Web APIs from 80 heterogenous online resources. In the following, we discuss our observations from performing these steps:

- *Expert discussions form a considerable proportion of the included resources*. As shown in Figure 1, expert discussions (i.e., books, design standards, weblogs and tutorials, and vendor white papers) form 41% of the included resources in this study. This large proportion of non-scholarly discussion may seem unconventional to the customary scientific literature reviews. However, this distribution of the selected resources arises from two factors: First, the research questions raised in this paper are related to the design techniques and their trade-offs. It is inherent to design techniques to be mostly introduced or discussed by practitioners. Thus, it is expected that expert discussions build a considerable proportion of the selected resources. Second, the distribution represents the availability of literature on the specific topic under study and suggests that scholarly discussions are still sparse on this topic. In such circumstances, expert discussions can not be filtered out since gathering, structuring, and analyzing expert discussions provides a substrate for extending scholarly discussions.
- *Scholarly discussions mainly focus on non-functional requirements while expert discussions mainly focus on design techniques*. *Both discussions address the trade-offs of the design techniques to a few extent using different approaches*. Out of the 47 included scholarly discussions (appearing the form of journal, conference, and workshop papers) 20 listed, defined, discussed, or evaluated one or several non-functional requirements, 25 analyzed or evaluated a design technique against one or several non-functional requirements, and only two listed a set of design techniques (The focus of the included research papers is identified in Appendix I). In contrast, all the 36 included expert discussions (appearing in the form of books, weblogs, vendor white-papers, tutorials, and design standards) mainly focused on listing, defining, or discussing one or several design techniques. Moreover, both scholarly and expert discussions sparsely addressed the trade-offs of one or several design techniques. The former mostly used formal analysis, proofs, or experiments to analyze or evaluate a design technique, while the latter only used anecdotal evidence and qualitative reasoning adopting some form of the template introduced for design patterns [32] either implicitly or explicitly.
- *API usability and API security are frequently discussed in scholarly discussions, and API security is frequently discussed in expert discussions. Other identified non-functional requirements have been barely discussed in both scholarly and expert discussions.* 59% of the 47 included scholarly discussions were either dedicated to API security or API usability: 17 resources merely focused on analysing and evaluating security of some design techniques, and 13 resources merely focused on defining, characterizing, or evaluating API usability, whereas other API non-functional requirements were sparsely addressed in the scholarly discussions. For example, only one resource discussed API visibility and only 4 papers were dedicated two API performance, while one of these papers were about the performance evaluation of an API security technique. Moreover, 32% of the 34 included expert discussions were merely dedicated to API security including (3 books, 6 design standards, and 2 whitepapers). These statistics suggest that very little attention has been given

to characterization and realization of other categories of the identified non-functional requirements in both scholarly and expert discussions.

- *Various terms, definitions, and characterizations are provided for API non-functional requirements in scholarly discussions.* While extracting non-functional requirements from the included resources, we noted that different resources define and characterize the same non-functional requirements differently. One prominent example is API Usability. Various resources(e.g., [8], [40], [44], [47], [51]) characterize this requirement differently and even studies are dedicated to gather and analyse all these definitions and characteristics [49]. To mitigate this variation when structuring non-functional requirements in this study, we have taken two actions: (1) we have considered the commonalities in various terms, definitions, and characterizations. (2) We have cross-checked the terms, definitions, and characterizations with ISO / IEC 25011 standard [33] and have used its definitions and classifications as a reference where applicable and have customized and extended them for web APIs. As a result of these two actions, the classification and definition of the non-functional requirements that has been provided in this study differ from both the included resources (the terms and the definitions are different from the included studies) and ISO / IEC 25011 standard (the terms, definitions, and classifications focus on Web APIs).

- *Evaluation and measurement of API non-functional requirements has received little attention in scholarly discussions and no attention in expert discussions*. Out of the nine categories of non-functional requirements identified in this study, we only found some scholarly resources on evaluating API usability [44], [45], [46], [47], [48], [49], [50], [51] and a few resources on evaluating API performance [37], [53] and API security [53]. We found no resource discussing the evaluation of the other six categories of identified non-functional requirements. These statistics suggest that little attention is given to measuring and evaluating non-functional requirements in APIs in both scholarly and expert discussions.

- *Qualitative reasoning and expert opinion is mostly the only type of available evidence supporting the trade-offs of the identified design techniques*. The only type of supporting evidence supporting for 19 out of the 22 design techniques for which we could identify some trade-offs is qualitative reasoning and expert opinion. This statistic suggests that very little attention is given to rigorous analysis and evaluation of web API design techniques against non-functional requirements both in scholarly and expert discussions.

- *Scholarly discussions only focus on the analyzing design frameworks against non-functional requirements.* 3 out of the 22 design techniques for which we could identify some trade-offs (namely OpenID Connect, Open Authorization, and Publish and Subscribe), were supported by some form of scientific evidence. However, these design techniques are rather design frameworks and can be implemented with various details. As emphasized in the related resources, two factors namely implementation details and the study settings influence the trade-offs the studied designs [17], [85], [86], [87], [88], [89], [98], [58], [100], [101], [102]

- *Little attention has been given to addressing non-functional requirements through design techniques and evaluating and analysing the trade-offs of design techniques against non-functional requirements in both scholarly and expert discussions.* In this study, we could identify nine groups of non-functional requirements, but could only associate design techniques to five groups of them. Moreover, we could identify 37 concrete design techniques, whereas we could pinpoint some trade-offs for 22 of these techniques. This decrease from non-functional requirements to design techniques, and from design techniques to trade-offs suggests that little attention has been given to the relations between design techniques and non-functional requirements in both scholarly and expert discussions.

7 Threats to Validity

Several precautions should be taken in referring to the design knowledge integrated into WEBAPIK. In the following, we discuss the threats to the reliability and validity of the presented design knowledge.

7.1 Threats to Internal Validity

Internal validity examines the extent to which the research steps performed to obtain the research results are free from bias and error [105]. In this following, we discuss how the collection, extraction and aggregation steps may have influenced the quality of the design knowledge compiled in WEBAPIK.

The collection of the knowledge resources could have been impacted by three factors: (a) the used search engines, (b) the period in which the resources were collected, and (c) the search process. Moreover, the organization of the design knowledge could have been influenced by the performed extraction and aggregation steps.

It is plausible that specific search engines or repositories do not index some resources or neglect showing the related resources in their top results based on the ranking algorithms they adopt. To minimize this influence, we have taken three steps: (1) We have used two web search engines of Google Scholar and Google and five popular Computer Science and Engineering databases of Web of Science, IEEE Xplore, ACM Digital Library, SpringerLink, and ScienceDirect. (2) We have performed the search and retrieval process in several rounds using the first set of extracted knowledge pieces as the input for the subsequent rounds of search to find further resources. (3) We have investigated the top three pages of the results produced by the search engines and repositories.

The date of resource collection is a threat to all systematic reviews and puts the comprehensiveness of the obtained knowledge from these reviews at risk. To reduce this risk, we have repeated the search and retrieval of the collected resources in two periods of March to August 2018 and August 2022. Thus, the resources related to the topic of this study that are made available after August 2022 are not included in this study. Nevertheless, we acknowledge that WEBAPIK is neither extensive nor comprehensive. Instead, it brings an initial structure to the design knowledge, provides evidence that the pieces of design knowledge related to this structure exist in scholarly and expert discussions, and shows that these pieces can be extracted and aggregated from various resource. Moreover, the initial structure brought to the design knowledge makes it amenable towards extension.

The retrieval and selection of the knowledge resources has been done manually. Thus, the researcher errors and judgement may have excluded some related resources from the collected and studied set. To mitigate the risk of human errors in resource collection and selection, we have taken two steps: (1) we systematized and elaborated the resource collection and selection steps to possible extent to reduce some biases and misinterpretations. (2) We have repeated these steps in several rounds in two different time periods.

The extraction and aggregation of the design knowledge from the selected resources is done manually. Hence, the researcher's understanding, judgement, interpretation, and decisions may have altered the intention of the original information in the related resources. Moreover, it is possible that invalid structures and relationships are introduced in non-functional requirements, design techniques or trade-offs. To mitigate this risk, we have systematized and elaborated the information extraction and aggregation steps to the extent possible to bring transparency to the performed steps. However, we acknowledge that incorporating multiple researchers to perform the knowledge collection and extraction steps would have addressed this risk more rigorously, although this action was impossible for us due to the limited resources allocated to this research.

7.2 Threats to External Validity

External validity examines the extent to which the results obtained from a research study can be generalized and applied to other contexts [105]. In the following, we discuss the reservations that threaten the reusability the design knowledge integrated into WEBAPIK.

Four issues threaten the rigor of the trade-offs collected in WEBAPIK: (1) Most of the presented trade-offs in WEBAPIK are extracted from the communicated advice and experience of one or two practitioners. This type of knowledge which is referred to as "expert opinion" or "anecdotal evidence" has low scientific validity and reliability. (2) The available expert opinion only includes qualitative reasoning and arguments to analyze the trade-offs of the design techniques. These arguments can be biased and may rise from the advocacy for a design technique, particularly in the resources where the authors analyse the design techniques that themselves have introduced. Moreover, since the arguments has no firm basis or strong supporting evidence, it is plausible that the opinion of different experts varies about the trade-offs of the same design techniques. (3) For most of the identified non-functional requirements, we could find no evaluation metrics to quantify the fulfilment of the requirements in a design technique to compare design alternatives. (4) Even in cases where evaluation studies exist, the evaluation of most of the non-functional requirements can be influenced by two study-specific factors: (a) the detailed implementation of a technique and the implementation settings, (b) the detailed setting of the experiments.

Nevertheless, it should be noted that all the enumerated weaknesses are inherent to the research questions raised in this study and mainly arise from the sparsity of scholarly discussions on the topic under study. First, design techniques and patterns are mainly introduced and analysed by experts. Hence, it is expected that expert opinion plays a significant role in the studies on design knowledge. Second, what can complement expert opinion is the availability of scientific studies and evidence which is absent due to the inadequacy of the scholarly discussions on the same topic.

8 Related Work

To develop WEBAPIK, we have systematically extracted, curated, and visualized knowledge about addressing non-functional requirements in the design of web APIs. Each of these steps has been already attempted in previous research efforts (e.g., [31], [106], [107], [108], [109], [110], [111]) to collect and structure the knowledge about addressing non-functional requirements in various areas of software design and development. The research reported in this paper complements the previous studies through systematizing, elaborating, and integrating all the knowledge extraction and curation steps, and applying these steps to structure the knowledge related to the unexplored domain of web API design. Moreover, this research furthers the existing literature on non-functional requirements by introducing one large and detailed body of structured knowledge about addressing non-functional requirements in a specific domain of software design.

Capturing and reusing software design knowledge is a long-studied topic in the field of software engineering and particularly in software architecture. In numerous studies, such as [32], [112], [113], [114], [115], [116], [117], the knowledge about various domains of software design has been structured in the form of design patterns. Various research efforts emphasize the need for reusing architectural knowledge, e.g., [118], [119], [120], [121], [122]. Also, several rational methodologies and concepts have been proposed, including Architecture Trade-Offs Analysis Method (ATAM) [123], Architecture Rationale and Element Linkage (AREL) [124], and Trade-off-oriented Development (ToF) [125]) that consider the knowledge of various design alternatives and their trade-offs in making architectural design decisions. Furthermore, a separate strand of research [126] discuss the use of various information and knowledge sources to assist developers with their development activities. WEBAPIK complements the above lines of research via linking the idea of reusing design and architectural knowledge to non-functional requirements and providing concrete knowledge support for making rational design decisions in the domain of web API design. Moreover, to develop WEBAPIK, we have exploited and structured the corpora of design knowledge available on the internet to provide a substrate for design decision making.

There are also a few attempts studying how to address or evaluate several non-functional requirements in the design of web APIs, e.g., [127] or collecting a body of API design techniques, e.g., [128], [129]. The research reported in this paper complements this strand of efforts through providing a detailed body of structured knowledge on non-functional requirements, design techniques to address these requirements and the possible trade-offs that should be made in selecting the identified design techniques.

9 Conclusions

Although dealing with non-functional requirements in software design has been long studied, there is still little guide on addressing these requirements in web APIs. To address this gap, in this paper, we presented WEBAPIK a body of structured knowledge on designing web APIs that contains 27 distinct non-functional requirements, 37 distinct design techniques to address some of the identified requirements, and the trade-offs of 22 design techniques. The knowledge compiled in WEBAPIK is extracted from 80 heterogenous online resources via performing a systematic and evidence-based literature review.

To the best of our knowledge, WEBAPIK is the first extensive study on addressing non-functional requirements in the domain of web APIs. Moreover, it furthers existing literature on non-functional requirements via structuring and compiling a large and detailed body of knowledge on defining and addressing these requirements in a specific domain of software design.

In WEBAPIK, the opinion and experience of practitioners as well as scholarly research and empirical evidence (in case available) are aggregated and structured to provide insight about addressing non-functional requirements in the design of web APIs. The collection and organization of the design knowledge in WEBAPIK paves the way for reusing the available information and knowledge scattered in numerous online resources. Moreover, the structure brought to the design knowledge facilitates the addition of new pieces of information and creates the potential to employ the organized knowledge in the recommender systems that aid software developers in design decision making.

The authors have already applied the graph representation of the knowledge presented in WEBAPIK to design and develop Rational API Designer (RAPID)[4], an open-source conversational and interactive knowledge-based assistant that provides guidelines for addressing non-functional requirements in the design of Web APIs [29],[130]. In RAPID, the presented knowledge graphs are formalized using a multi-valued logic and are utilized in a stepwise logical inference and evaluation procedure to transform textual requirements into concrete design fragments.

We envision five future research directions based on the outcomes of this study:

- Furthering research on evaluating the identified non-functional requirements via identifying qualitative and quantitative metrics.
- Strengthening the evidence to support or refute the presented trade-offs through conducting various forms of empirical studies, including case reports, case studies, and controlled experiments run.
- Examining the validity, thoroughness, and reliability of the structured design knowledge in further studies.
- Extending and refining the body of WEBAPIK via structuring and adding further pieces of design knowledge or modifying the presented knowledge.
- Validating and improving the resource retrieval and knowledge extraction and aggregation procedures via applying the performed steps to collect and compile the knowledge of other design domains as well as automating these steps to reduce both the time allocated to the study and the human biases, and errors introduced in the outcomes.

10 Acknowledgement

The first author would like to thank Prof. Marsha Chechik, and Prof. Steve Easterbrook at the University of Toronto for their supervision throughout the course of the research reported in this paper. She also thanks Prof. John Mylopoulos for providing feedback.

11 References

1. Jansen, S., Finkelstein, A., & Brinkkemper, S. (2009). A sense of community: A research agenda for software ecosystems. In Proceedings of 31st International Conference on Software Engineering - Companion Volume (pp. 187-190). IEEE.

[4] https://github.com/m-h-s/RAPID

2. Sadi, M. H., & Yu, E. (2014). Analyzing the evolution of software development: From creative chaos to software ecosystems. In eighth international conference on research challenges in information science (RCIS) (pp. 1-11). IEEE.

3. Tan, L., & Wang, N. (2010). Future internet: The internet of things. In 2010 3rd international conference on advanced computer theory and engineering (ICACTE) (Vol. 5, pp. V5-376). IEEE.

4. Bosch, J. (2010). Architecture challenges for software ecosystems. In Proceedings of the Fourth European Conference on Software Architecture: Companion Volume (pp. 93-95). ACM.

5. Vukovic, M., Laredo, J., & Rajagopal, S. (2014). API Terms and Conditions as a Service. In 2014 IEEE International Conference on Services Computing (pp. 386-393). IEEE.

6. Weber, R. H. (2010). Internet of Things–New security and privacy challenges. Computer law & security review, 26(1), 23-30.

7. Sicari, S., Rizzardi, A., Grieco, L. A., & Coen-Porisini, A. (2015). Security, privacy, and trust in Internet of Things: The road ahead. Computer networks, 76, 146-164.

8. Stylos, J., & Myers, B. (2007). Mapping the space of API design decisions. In Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on (pp. 50-60). IEEE.

9. Myers, B. A., & Stylos, J. (2016). Improving API usability. Communications of the ACM, 59(6), 62-69.

10. Siriwardena, P. (2014). Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE. Apress, Berkeley, CA.

11. De, B. (2017). API Management: An Architect's Guide to Developing and Managing APIs for Your Organization. Apress, Berkeley, CA, First edition March 2017.

12. Vijayakumar, T. (2018). Practical API Architecture and Development with Azure and AWS. Apress, Berkeley, CA.

13. Madden, N. (2020). API security in action. Manning Publications.

14. Richardson, C. Pattern: API Gateway. Backend for Front-End, 37-40, Available at http://microservices.io/patterns/apigateway.html.

15. RFC 6749 (2012). The OAuth 2.0 Authorization Framework, Available at https://www.rfc-editor.org/rfc/rfc6749.

16. Sakimura, N., Bradley, D., de Mederiso, B., Jones, M., & Jay, E. (2012). OpenID connect standard 1.0-draft 09, Available at https://openid.net/specs/openid-connect-standard-1_0-09.html.

17. Sun, S. T., & Beznosov, K. (2012). The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In Proceedings of the 2012 ACM conference on Computer and communications security (pp. 378-390). ACM.

18. Li, W., & Mitchell, C. J. (2016). Analyzing the security of Google's implementation of OpenID Connect. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (pp. 357-376). Springer, Cham.

19. Cataldo, M., & Herbsleb, J. D. (2010). Architecting in software ecosystems: interface translucence as an enabler for scalable collaboration. In Proceedings of the Fourth European Conference on Software Architecture: Companion Volume (pp. 65-72). ACM.

20. Bloch, J. (2006). How to design a good API and why it matters. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (pp. 506-507). ACM.

21. Henning, M. (2009). API design matters. Communications of the ACM, 52(5), 46-56.

22. Kitchenham, B. (2004). Procedures for performing systematic reviews. Keele, UK, Keele University, Technical Report TR/SE-0401, 1-26.

23. Dyba, T., Kitchenham, B. A., & Jorgensen, M. (2005). Evidence-based software engineering for practitioners. IEEE software, 22(1), 58-65.

24. Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In Proceedings of the 18th international conference on evaluation and assessment in software engineering (pp. 1-10).

25. Flick, U. (2009). An introduction to qualitative research. Sage Publications Limited.

26. Saldaña, J. (2009). The coding manual for qualitative researchers. Sage Publications Limited.

27. Thomas, D. R. (2006). A general inductive approach for analyzing qualitative evaluation data. American journal of evaluation, 27(2), 237-246.

28. Hogan, A:, Blomqvist, E:, Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Labra Gayo, J. E., Kirrane, S., Neumaier, S., Polleres, A., Navigli, R., Ngonga Ngomo, A:-C., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., & Zimmermann, A. (2020). Knowledge graphs. ACM Computing Surveys (CSUR), 54(4), 1-37.

29. Sadi, M. H. (2020). Assisting with API Design Through Reusing Design Knowledge (Doctoral dissertation, University of Toronto (Canada)).

30. Pillai, S., Iijima, K., O'Neill, M., Santoro, J., Jain, A., Ryan, F., & (2021). Magic Quadrant for Full Life Cycle API Management. The Gartner Group.

31. Chung, L., Nixon, B. A., Yu, E., & Mylopoulos, J. (2000). Non-functional requirements in software engineering (Vol. 5). Springer Science & Business Media.

32. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In European Conference on Object-Oriented Programming (pp. 406-431).

33. ISO / IEC TS 25011: 2017 Information technology — Systems and Software Quality Requirements and Evaluation (SQuaRE) — Service quality models, Available at: https://www.iso.org/obp/ui#iso:std:iso-iec:ts:25011:ed-1:v2:en.

34. Bass, L., Clements, P., & Kazman, R. (2003). Software architecture in practice. Addison-Wesley Professional.

35. Kruchten, P. B. (1995). The 4+1 view model of architecture. IEEE software, 12(6), 42-50.

36. Akana Documents, How to Accelerate API adoption – Available at https://www.akana.com/blog/api-adoption on 2019-08-13.

37. Bermbach, D., & Wittern, E. (2016). Benchmarking web api quality. In International Conference on Web Engineering (pp. 188-206). Springer, Cham.

38. Zghidi, A., Hammouda, I., Hnich, B., & Knauss, E. (2017). On the role of fitness dimensions in API design assessment-an empirical investigation. In 2017 IEEE/ACM 1st International Workshop on API Usage and Evolution (WAPI) (pp. 19-22). IEEE.

39. Richardson, C. Building Micro-Services: Inter-Process Communication in a Micro-Service Architecture, Available at https://www.nginx.com/blog/building-microservices-inter-process-communication/

40. McLellan, S. G., Roesler, A. W., Tempest, J. T., & Spinuzzi, C. I. (1998). Building more usable APIs. IEEE software, 15(3), 78-86.

41. Robillard, M. P. (2009). What makes APIs hard to learn? Answers from developers. IEEE software, 26(6), 27-34.

42. Robillard, M. P., & Deline, R. (2011). A field study of API learning obstacles. Empirical Software Engineering, 16(6), 703-732.

43. Piccioni, M., Furia, C. A., & Meyer, B. (2013). An empirical study of API usability. In Empirical Software Engineering and Measurement, 2013 ACM/IEEE international symposium on (pp. 5-14). IEEE.

44. Zibran, M. F., Eishita, F. Z., & Roy, C. K. (2011). Useful, but usable? factors affecting the usability of APIs. In 18th Working Conference on Reverse Engineering (WCRE), 2011 (pp. 151-155). IEEE.

45. Scheller, T., & Kühn, E. (2015). Automated measurement of API usability: The API concepts framework. Information and Software Technology, 61, 145-162.

46. Koçi, R., Franch, X., Jovanovic, P., & Abelló, A. (2020). A data-driven approach to measure the usability of web APIs. In 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 64-71). IEEE.

47. Bore, C., & Bore, S. (2005). Profiling software API usability for consumer electronics. In 2005 Digest of Technical Papers. International Conference on Consumer Electronics, 2005. ICCE. (pp. 155-156). IEEE.

48. Rama, G. M., & Kak, A. (2015). Some structural measures of API usability. Software: Practice and Experience, 45(1), 75-110.

49. Rauf, I., Troubitsyna, E., & Porres, I. (2019). Systematic mapping study of API usability evaluation methods. Computer Science Review, 33, 49–68.

50. Mosqueira-Rey, E., Alonso-Ríos, D., Moret-Bonillo, V., Fernández-Varela, I., & Álvarez-Estévez, D. (2018). A systematic approach to API usability: Taxonomy-derived criteria and a case study. Information and Software Technology, 97, 46-63.

51. Grill, T., Polacek, O., & Tscheligi, M. (2012). Methods towards API usability: A structural analysis of usability problem categories. In International conference on human-centered software engineering (pp. 164-180). Springer, Berlin, Heidelberg.

52. Xu, J., Wang, Y., Chen, P., & Wang, P. (2017). Lightweight and adaptive service api performance monitoring in highly dynamic cloud environment. In 2017 IEEE International Conference on Services Computing (SCC) (pp. 35-43). IEEE.

53. Bermbach, D., & Wittern, E. (2019). Benchmarking Web API Quality--Revisited. arXiv preprint arXiv:1903.07712.

54. Adeborna, E., & Fletcher, K. K. (2020). An empirical study of web api quality formulation. In International Conference on Services Computing (pp. 145-153). Springer, Cham.

55. MuleSoft. Guide to API Security, Available at https://www.mulesoft.com/resources/api/api-security. Retrieved on 2022 – 08 – 15.

56. Villanueva, J. C. Comparing Load Balancing Algorithms. Available at https://www.jscape.com/blog/load-balancing-algorithms.

57. Kemp Technologies White Paper. Load Balancing Algorithms and Techniques, Available at https://kemptechnologies.com/load-balancer/load-balancing-algorithms-techniques/.

58. Microsoft Documents. Caching, Available at https://docs.microsoft.com/en-us/azure/architecture/best-practices/caching.

59. Richardson, C. Building Micro-services: Using an API Gateway, Available at https://www.nginx.com/blog/building-microservices-using-an-api-gateway/.

60. Fowler, M. (2002). Patterns of Enterprise Architecture Applications, Addison-Wesley Professional; First edition (Nov. 5 2002).

61. Fowler, M. Gateway Pattern, Available at https://martinfowler.com/eaaCatalog/gateway.html.

62. Richardson, C. Service Discovery in a Micro-Service Architecture. Available at https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/.

63. Richardson, C. Pattern: Self Registration. Available at https://microservices.io/patterns/self-registration.html.

64. Richardson, C. Pattern: 3rd Party Registration. Available at https://microservices.io/patterns/3rd-party-registration.html.

65. Dey, S. & Mulloy, B. (2012). Essential Facade Patterns – API Composition. Available at https://www.slideshare.net/apigee/api-facade-patterns-composition.

66. Richardson, C (2017). Pattern: API Composition, Available at https://microservices.io/patterns/data/api-composition.html. Retrieved on 2022 – 08- 10.

67. Dey, S. (2012). Essential API Facade Patterns – Synchronous to Asynchronous Conversion. Available at https://www.slideshare.net/apigee/essential-api-facade-patterns-synchronous-to-asynchronous-conversion-episode-4.

68. Richardson, C. Pattern: Server-Side Service Discovery. https://microservices.io/patterns/server-side-discovery.html.

69. Richardson, C. Pattern: Client-Side Service Discovery. Available at https://microservices.io/patterns/client-side-discovery.html https://microservices.io/patterns/client-side-discovery.html.

70. Hohpe, G., & Woolf, B. Enterprise Integration Patterns. Available at https://www.enterpriseintegrationpatterns.com/patterns/messaging/index.html.

71. Hohpe, G., & Woolf, B. (2004). Enterprise integration patterns: Designing, building, and deploying messaging solutions. Addison-Wesley Professional.

72. RFC 4158: Internet X.509, Public Key Infrastructure: Certification Path Building, Available at https://tools.ietf.org/html/rfc4158.

73. RFC 5280, Internet X.509, Public Key Infrastructure and Certificate Revocation List, Available at https://www.rfc-editor.org/rfc/rfc3280.

74. OAuth 2.0, Available at https://oauth.net/2/.

75. OpenID Connect, Available at https://openid.net/connect/.

76. Google Cloud. Why and when to use API keys, Available at https://cloud.google.com/endpoints/docs/openapi/when-why-api-key.

77. Stocker, M., Zimmermann, O., Zdun, U., Lübke, D., & Pautasso, C. (2018). Interface quality patterns: Communicating and improving the quality of microservices Apis. In Proceedings of the 23rd European Conference on Pattern Languages of Programs (pp. 1-16).

78. Tang, L., Ouyang, L., & Tsai, W. T. (2015). Multi-factor web API security for securing Mobile Cloud. In 2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD) (pp. 2163-2168). IEEE.

79. Fowler, M. Circuit Breaker, Available at https://martinfowler.com/bliki/CircuitBreaker.html..

80. Montesi, F., & Weber, J. (2016). Circuit breakers, discovery, and API gateways in microservices. arXiv preprint-arXiv:1609.05830.

81. Apigee Reference Material. Comparing Quota, Spike Arrest, and Concurrent Rate Limit Policies, Available at https://docs.apigee.com/api-platform/develop/comparing-quota-spike-arrest-and-concurrent-rate-limit-policies.

82. Wilson, Y., & Hingnikar, A. (2019). Solving Identity Management in Modern Applications: Demystifying OAuth 2.0, OpenID Connect, and SAML 2.0. Apress.

83. Fett, D., Küsters, R., & Schmitz, G. (2016). A comprehensive formal security analysis of OAuth 2.0. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (pp. 1204-1215).

84. Yang, F., & Manoharan, S. (2013). A security analysis of the OAuth protocol. In 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM) (pp. 271-276). IEEE.

85. Li, W., & Mitchell, C. J. (2014). Security issues in OAuth 2.0 SSO implementations. In International Conference on Information Security (pp. 529-541). Springer, Cham.

86. Ferry, E., Raw, J. O., & Curran, K. (2015). Security evaluation of the OAuth 2.0 framework. Information & Computer Security.

87. Darwish, M., & Ouda, A. (2015). Evaluation of an OAuth 2.0 protocol implementation for web server applications. In 2015 International Conference and Workshop on Computing and Communication (IEMCON) (pp. 1-4). IEEE.

88. Singh, J., & Chaudhary, N. K. (2022). OAuth 2.0: Architectural design augmentation for mitigation of common security vulnerabilities. Journal of Information Security and Applications, 65, 103091.

89. Blazquez, A., Tsiatsis, V., & Vandikas, K. (2015). Performance evaluation of OpenID Connect for an IOT information market-place. In Vehicular Technology Conference (VTC Spring), 2015 IEEE 81st (pp. 1-6). IEEE.

90. Hammann, S., Sasse, R., & Basin, D. (2020). Privacy-preserving openid connect. In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (pp. 277-289).

91. Li, W., & Mitchell, C. J. (2020). User access privacy in OAuth 2.0 and OpenID connect. In 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) (pp. 664-6732). IEEE.

92. Mainka, C., Mladenov, V., Schwenk, J., & Wich, T. (2017). Sok: Single sign-on security—an evaluation of openid connect. In 2017 IEEE European Symposium on Security and Privacy (EuroS&P) (pp. 251-266). IEEE.

93. Fett, D., Küsters, R., & Schmitz, G. (2017). The web sso standard openid connect: In-depth formal security analysis and security guidelines. In 2017 IEEE 30th Computer Security Foundations Symposium (CSF) (pp. 189-202). IEEE.

94. Navas, J., & Beltrán, M. (2019). Understanding and mitigating OpenID Connect threats. Computers & Security, 84, 1-16.

95. Li, W., Mitchell, C. J., & Chen, T. (2019). Oauthguard: Protecting user security and privacy with oauth 2.0 and openid connect. In Proceedings of the 5th ACM Workshop on Security Standardisation Research Workshop (pp. 35-44).

96. Mladenov, V., Mainka, C., & Schwenk, J. (2015). On the security of modern single sign-on protocols: Second-order vulnerabilities in openid connect. arXiv preprint arXiv:1508.04324.

97. Behnel, S., Fiege, L., & Muhl, G. (2006). On quality-of-service and publish-subscribe. In 26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06) (pp. 20-20). IEEE.

98. Cugola, G., Margara, A., & Migliavacca, M. (2009). Context-aware publish-subscribe: Model, implementation, and evaluation. In 2009 IEEE symposium on computers and communications (pp. 875-881). IEEE.

99. Costa, P., Migliavacca, M., Picco, G. P., & Cugola, G. (2004). Epidemic algorithms for reliable content-based publish-subscribe: An evaluation. In 24th International Conference on Distributed Computing Systems, 2004. Proceedings. (pp. 552-561). IEEE.

100. Lazidis, A., Tsakos, K., & Petrakis, E. G. (2022). Publish–Subscribe approaches for the IoT and the cloud: Functional and performance evaluation of open-source systems. Internet of Things, 19, 100538.

101. Oh, S., Kim, J. H., & Fox, G. (2010). Real-time performance analysis for publish/subscribe systems. Future Generation Computer Systems, 26(3), 318-323.

102. Wardana, A. A., & Perdana, R. S. (2018). Access control on internet of things based on publish/subscribe using authentication server and secure protocol. In 2018 10th International Conference on Information Technology and Electrical Engineering (ICITEE) (pp. 118-123). IEEE.

103. Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural patterns for microservices: a systematic mapping study. In CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018. SciTePress.

104. Tighilt, R., Abdellatif, M., Moha, N., Mili, H., Boussaidi, G. E., Privat, J., & Guéhéneuc, Y. G. (2020,). On the study of microservices antipatterns: A catalog proposal. In Proceedings of the European Conference on Pattern Languages of Programs 2020 (pp. 1-13).

105. Siegmund, J., Siegmund, N., & Apel, S. (2015). Views on internal and external validity in empirical software engineering. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (Vol. 1, pp. 9-19). IEEE.

106. Chung, L., & Supakkul, S. (2006). Capturing and reusing functional and non-functional requirements knowledge: a goal-object pattern approach. In 2006 IEEE International Conference on Information Reuse & Integration (pp. 539-544). IEEE.

107. Casamayor, A., Godoy, D., & Campo, M. (2010). Identification of non-functional requirements in textual specifications: A semi-supervised learning approach. Information and Software Technology, 52(4), 436-445.

108. Supakkul, S., & Chung, L. (2010). Visualizing non-functional requirements patterns. In 2010 Fifth International Workshop on Requirements Engineering Visualization (pp. 25-34). IEEE.

109. Sadi, M. H., & Yu, E. (2017). Modeling and analyzing openness trade-offs in software platforms: a goal-oriented approach. In International Working Conference on Requirements Engineering: Foundation for Software Quality (pp. 33-49). Springer, Cham.

110. Sadi, M. H., & Yu, E. (2017). Accommodating openness requirements in software platforms: a goal-oriented approach. In International Conference on Advanced Information Systems Engineering (pp. 44-59). Springer, Cham.

111. Binkhonain, M., & Zhao, L. (2019). A review of machine learning algorithms for identification and classification of non-functional requirements. Expert Systems with Applications: X, 1, 100001.

112. Buschmann, F., Henney, K., & Schmidt, D. C. (2007). Pattern-oriented software architecture, on patterns and pattern languages (Vol. 5). John wiley & sons.

113. Aridor, Y., & Lange, D. B. (1998). Agent design patterns: Elements of agent application design. In Proceedings of the second international conference on Autonomous agents (pp. 108-115).

114. Freeman, E., Robson, E., Bates, B., & Sierra, K. (2008). Headfirst design patterns. " O'Reilly Media, Inc.".

115. Erl, T. (2008). SOA Design Patterns (paperback). Pearson Education.

116. Heer, J., & Agrawala, M. (2006). Software design patterns for information visualization. IEEE transactions on visualization and computer graphics, 12(5), 853-860.

117. Zhang, C., & Budgen, D. (2011). What do we know about the effectiveness of software design patterns?. IEEE Transactions on Software Engineering, 38(5), 1213-1231.

118. Boehm, B., & In, H. (1996). Identifying quality-requirement conflicts. IEEE software, 13(2), 25-35.

119. Monroe, R. T., Kompanek, A., Melton, R., & Garlan, D. (1997). Architectural styles, design patterns, and objects. IEEE software, 14(1), 43-52.

120. Babar, M. A., Gorton, I., & Jeffery, R. (2005). Capturing and using software architecture knowledge for architecture-based software development. In Fifth International Conference on Quality Software (QSIC'05) (pp. 169-176). IEEE.

121. Farshidi, S., Jansen, S., & van der Werf, J. M. (2020). Capturing software architecture knowledge for pattern-driven design. Journal of Systems and Software, 169, 110714.

122. Kruchten, P. (2010). Where did all this good architectural knowledge go?. In European Conference on Software Architecture (pp. 5-6). Springer, Berlin, Heidelberg.

123. Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., & Carriere, J. (1998). The architecture trade-off analysis method. In Engineering of Complex Computer Systems, 1998. ICECCS'98. Proceedings. Fourth IEEE International Conference on (pp. 68-78). IEEE.

124. Tang, A., Jin, Y., & Han, J. (2007). A rationale-based architecture model for design traceability and reasoning. Journal of Sys-tems and Software, 80(6), 918-934.

125. Dürschmid, T., Kang, E., & Garlan, D. (2019). Trade-off-oriented development: making quality attribute trade-offs first-class. In 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSENIER) (pp. 109-112). IEEE.

126. Robillard, M., Walker, R., & Zimmermann, T. (2009). Recommendation systems for software engineering. IEEE software, 27(4), 80-86.

127. Costa, B., Pires, P. F., Delicato, F. C., & Merson, P. (2016). Evaluating REST architectures—Approach, tooling and guidelines. Journal of Systems and Software, 112, 156-180.

128. Mathijssen, M., Overeem, M., & Jansen, S. (2020). Identification of practices and capabilities in API management: a systematic literature review. arXiv preprint arXiv:2006.10481.

129. Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C., & Zdun, U. (2019). Introduction to microservice API patterns (MAP).

130. Sadi, M. H., & Yu, E. (2021). RAPID: a knowledge-based assistant for designing web APIs. Requirements Engineering, 1-52.

Appendix I: The Classification of the Collected Knowledge Resources

The knowledge resources used to develop WEBAPIK are listed in Table 1.

The resources are classified based on their type into six categories: B: Book, WL: Weblog, T: Tutorial, WP: White Paper, S: Standard Framework, and R: Research Paper.

The resources are classified based on their focus of study into three categories: DES-TECH: discussing design techniques, NFR: discussing non-functional requirements, EFFECT: discussing trade-offs.

Table 1 The Classified Knowledge Resources

	Type	Focus	Reference
1	B1	DES-TECH	De, B. (2017). API Management: An Architect's Guide to Developing and Managing APIs for Your Organization. Apress, Berkeley, CA, First edition March 2017.
2	B2	DES-TECH	Vijayakumar, T. (2018). Practical API Architecture and Development with Azure and AWS. Apress, Berkeley, CA.
3	B3	DES-TECH -Security	Madden, N. (2020). API security in action. Manning Publications.
4	B4	DES-TECH -Security	Wilson, Y., & Hingnikar, A. (2019). Solving Identity Management in Modern Applications: Demystifying OAuth 2.0, OpenID Connect, and SAML 2.0. Apress.
5	B5	DES-TECH -Security	Siriwardena, P. (2014). Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE. Apress, Berkeley, CA.
6	B6	DES-TECH	Hohpe, G., & Woolf, B. (2004). Enterprise integration patterns: Designing, building, and deploying messaging solutions. Addison-Wesley Professional.
7	B7	DES-TECH	Fowler, M. (2002). Patterns of Enterprise Architecture Applications, Addison-Wesley Professional; First edition (Nov. 5 2002).
8	WL1	DES-TECH	Richardson, C. Building Micro-Services: Inter-Process Communication in a Micro-Service Architecture, Available at https://www.nginx.com/blog/building-microservices-inter-process-communication/.
9	WL2	DES-TECH	Richardson, C. Building Micro-services: Using an API Gateway, Available at https://www.nginx.com/blog/building-microservices-using-an-api-gateway/.
10	WL3	DES-TECH	Richardson, C. Pattern: API Gateway. Backend for Front-End, 37-40, Available at http://microservices.io/patterns/apigateway.html.

11	WL4	DES-TECH	Richardson, C. Service Discovery in a Micro-Service Architecture. Available at https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/.
12	WL5	DES-TECH	Richardson, C. Pattern: Self Registration. Available at https://microservices.io/patterns/self-registration.html.
13	WL6	DES-TECH	Richardson, C. Pattern: 3rd Party Registration. Available at https://microservices.io/patterns/3rd-party-registration.html.
14	WL7	DES-TECH	Richardson, C. Pattern: API Composition, Available at https://microservices.io/patterns/data/api-composition.html.
15	WL8	DES-TECH	Richardson, C. Pattern: Server-Side Service Discovery. https://microservices.io/patterns/server-side-discovery.html.
16	WL9	DES-TECH	Richardson, C. Pattern: Client-Side Service Discovery. Available at https://microservices.io/patterns/client-side-discovery.html https://microservices.io/patterns/client-side-discovery.html.
17	WL10	DES-TECH	Fowler, M. Circuit Breaker, Available at https://martinfowler.com/bliki/CircuitBreaker.html.
18	WL11	DES-TECH	Fowler, M. Gateway Pattern, Available at https://martinfowler.com/eaaCatalog/gateway.html.
19	WL12	DES-TECH	Hohpe, G., & Woolf, B. Enterprise Integration Patterns. Available at https://www.enterpriseintegrationpatterns.com/patterns/messaging/index.html.
20	WL13	DES-TECH	Villanueva, J. C. Comparing Load Balancing Algorithms. Available at https://www.jscape.com/blog/load-balancing-algorithms.
21	T1	DES-TECH	Dey, S. & Mulloy, B. (2012). Essential Facade Patterns – API Composition. Available at https://www.slideshare.net/apigee/api-facade-patterns-composition.
22	T2	DES-TECH	Dey, S. (2012). Essential API Facade Patterns – Synchronous to Asynchronous Conversion. Available at https://www.slideshare.net/apigee/essential-api-facade-patterns-synchronous-to-asynchronous-conversion-episode-4.
23	WP1	DES-TECH	Apigee Reference Material. Comparing Quota, Spike Arrest, and Concurrent Rate Limit Policies, Available at https://docs.apigee.com/api-platform/develop/comparing-quota-spike-arrest-and-concurrent-rate-limit-policies.
24	WP2	DES-TECH	Kemp Technologies White Paper. Load Balancing Algorithms and Techniques, Available at https://kemptechnologies.com/load-balancer/load-balancing-algorithms-techniques/.
25	WP3	DES-TECH	Microsoft Documents. Caching, Available at https://docs.microsoft.com/en-us/azure/architecture/best-practices/caching.

26	WP4	DES-TECH -Security	Google Cloud. Why and when to use API keys, Available at https://cloud.google.com/endpoints/docs/openapi/when-why-api-key.
27	WP5	DES-TECH -Security	MuleSoft. Guide to API Security, Available at https://www.mulesoft.com/resources/api/api-security.
28	S1	DES-TECH -Security	RFC 6749 (2012). The OAuth 2.0 Authorization Framework, Available at https://www.rfc-editor.org/rfc/rfc6749.
29	S2	DES-TECH -Security	OAuth 2.0, Available at https://oauth.net/2/.
30	S3	DES-TECH -Security	Sakimura, N., Bradley, D., de Mederiso, B., Jones, M., & Jay, E. (2012). OpenID connect standard 1.0-draft 09, Available at https://openid.net/specs/openid-connect-standard-1_0-09.html.
31	S4	DES-TECH -Security	OpenID Connect, Available at https://openid.net/connect/.
32	S5	DES-TECH -Security	RFC 4158: Internet X.509, Public Key Infrastructure: Certification Path Building, Available at https://tools.ietf.org/html/rfc4158.
33	S6	DES-TECH -Security	RFC 5280, Internet X.509, Public Key Infrastructure and Certificate Revocation List, Available at https://www.rfc-editor.org/rfc/rfc3280
34	R1	NFR	Stylos, J., & Myers, B. (2007). Mapping the space of API design decisions. In Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on (pp. 50-60). IEEE.
35	R2	NFR -Usability	Myers, B. A., & Stylos, J. (2016). Improving API usability. Communications of the ACM, 59(6), 62-69.
36	R3	NFR -Visibility	Cataldo, M., & Herbsleb, J. D. (2010). Architecting in software ecosystems: interface translucence as an enabler for scalable collaboration. In Proceedings of the Fourth European Conference on Software Architecture: Companion Volume (pp. 65-72). ACM.
37	R4	NFR -Usability	McLellan, S. G., Roesler, A. W., Tempest, J. T., & Spinuzzi, C. I. (1998). Building more usable APIs. IEEE software, 15(3), 78-86.
38	R5	NFR -Usability	Robillard, M. P. (2009). What makes APIs hard to learn? Answers from developers. IEEE software, 26(6), 27-34.
39	R6	NFR -Usability	Robillard, M. P., & Deline, R. (2011). A field study of API learning obstacles. Empirical Software Engineering, 16(6), 703-732.
40	R7	NFR -Usability	Piccioni, M., Furia, C. A., & Meyer, B. (2013). An empirical study of API usability. In Empirical Software Engineering and Measurement, 2013 ACM/IEEE international symposium on (pp. 5-14). IEEE.

41	R8	NFR -Usability	Zibran, M. F., Eishita, F. Z., & Roy, C. K. (2011). Useful, but usable? factors affecting the usability of APIs. In 18th Working Conference on Reverse Engineering (WCRE), 2011 (pp. 151-155). IEEE.
42	R9	DES-TECH	Montesi, F., & Weber, J. (2016). Circuit breakers, discovery, and API gateways in microservices. arXiv preprintarXiv:1609.05830.
43	R10	NFR -Usability	Bore, C., & Bore, S. (2005). Profiling software API usability for consumer electronics. In 2005 Digest of Technical Papers. International Conference on Consumer Electronics, 2005. ICCE. (pp. 155-156). IEEE.
44	R11	NFR	Bermbach, D., & Wittern, E. (2016). Benchmarking web api quality. In International Conference on Web Engineering (pp. 188-206). Springer, Cham.
45	R12	NFR	Bermbach, D., & Wittern, E. (2019). Benchmarking Web API Quality--Revisited. arXiv preprint arXiv:1903.07712.
46	R13	NFR	Adeborna, E., & Fletcher, K. K. (2020). An empirical study of web api quality formulation. In International Conference on Services Computing (pp. 145-153). Springer, Cham.
47	R14	NFR	Zghidi, A., Hammouda, I., Hnich, B., & Knauss, E. (2017). On the role of fitness dimensions in API design assessment-an empirical investigation. In 2017 IEEE/ACM 1st International Workshop on API Usage and Evolution (WAPI) (pp. 19-22). IEEE.
48	R15	NFR -Usability	Koçi, R., Franch, X., Jovanovic, P., & Abelló, A. (2020). A data-driven approach to measure the usability of web APIs. In 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 64-71). IEEE.
49	R16	NFR -Usability	Rauf, I., Troubitsyna, E., & Porres, I. (2019). Systematic mapping study of API usability evaluation methods. Computer Science Review, 33, 49–68.
50	R17	NFR -Usability	Scheller, T., & Kühn, E. (2015). Automated measurement of API usability: The API concepts framework. Information and Software Technology, 61, 145-162.
51	R18	NFR -Usability	Rama, G. M., & Kak, A. (2015). Some structural measures of API usability. Software: Practice and Experience, 45(1), 75-110.
52	R19	NFR -Usability	Mosqueira-Rey, E., Alonso-Ríos, D., Moret-Bonillo, V., Fernández-Varela, I., & Álvarez-Estévez, D. (2018). A systematic approach to API usability: Taxonomy-derived criteria and a case study. Information and Software Technology, 97, 46-63.
53	R20	NFR -Usability	Grill, T., Polacek, O., & Tscheligi, M. (2012). Methods towards API usability: A structural analysis of usability problem categories. In International conference on human-centered software engineering (pp. 164-180). Springer, Berlin, Heidelberg.
54	R21	NFR -Performance	Xu, J., Wang, Y., Chen, P., & Wang, P. (2017). Lightweight and adaptive service api performance monitoring in highly dynamic cloud environment. In 2017 IEEE International Conference on Services Computing (SCC) (pp. 35-43). IEEE.

55	R22	EFFECT	Stocker, M., Zimmermann, O., Zdun, U., Lübke, D., & Pautasso, C. (2018). Interface quality patterns: Communicating and improving the quality of microservices Apis. In Proceedings of the 23rd European Conference on Pattern Languages of Programs (pp. 1-16).
56	R23	DES-TECH	Tang, L., Ouyang, L., & Tsai, W. T. (2015). Multi-factor web API security for securing Mobile Cloud. In 2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD) (pp. 2163-2168). IEEE.
57	R24	EFFECT -Security	Sun, S. T., & Beznosov, K. (2012). The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In Proceedings of the 2012 ACM conference on Computer and communications security (pp. 378-390). ACM.
58	R25	EFFECT -Security	Fett, D., Küsters, R., & Schmitz, G. (2016). A comprehensive formal security analysis of OAuth 2.0. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (pp. 1204-1215).
59	R26	EFFECT -Security	Yang, F., & Manoharan, S. (2013). A security analysis of the OAuth protocol. In 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM) (pp. 271-276). IEEE.
60	R27	EFFECT -Security	Li, W., & Mitchell, C. J. (2014). Security issues in OAuth 2.0 SSO implementations. In International Conference on Information Security (pp. 529-541). Springer, Cham.
61	R28	EFFECT -Security	Ferry, E., Raw, J. O., & Curran, K. (2015). Security evaluation of the OAuth 2.0 framework. Information & Computer Security.
62	R29	EFFECT -Security	Darwish, M., & Ouda, A. (2015). Evaluation of an OAuth 2.0 protocol implementation for web server applications. In 2015 International Conference and Workshop on Computing and Communication (IEMCON) (pp. 1-4). IEEE.
63	R30	EFFECT -Security	Singh, J., & Chaudhary, N. K. (2022). OAuth 2.0: Architectural design augmentation for mitigation of common security vulnerabilities. Journal of Information Security and Applications, 65, 103091.
64	R31	EFFECT -Security	Li, W., & Mitchell, C. J. (2016). Analyzing the security of Google's implementation of OpenID Connect. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (pp. 357-376). Springer, Cham.
65	R32	EFFECT -Security	Hammann, S., Sasse, R., & Basin, D. (2020). Privacy-preserving openid connect. In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (pp. 277-289).
66	R33	EFFECT -Performance	Blazquez, A., Tsiatsis, V., & Vandikas, K. (2015). Performance evaluation of OpenID Connect for an IOT information market-place. In Vehicular Technology Conference (VTC Spring), 2015 IEEE 81st (pp. 1-6). IEEE.

67	R34	EFFECT -Security	Li, W., & Mitchell, C. J. (2020). User access privacy in OAuth 2.0 and OpenID connect. In 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) (pp. 664-6732). IEEE.
68	R35	EFFECT -Security	Mainka, C., Mladenov, V., Schwenk, J., & Wich, T. (2017). Sok: Single sign-on security—an evaluation of openid connect. In 2017 IEEE European Symposium on Security and Privacy (EuroS&P) (pp. 251-266). IEEE.
69	R36	EFFECT -Security	Fett, D., Küsters, R., & Schmitz, G. (2017). The web sso standard openid connect: In-depth formal security analysis and security guidelines. In 2017 IEEE 30th Computer Security Foundations Symposium (CSF) (pp. 189-202). IEEE.
70	R37	EFFECT -Security	Navas, J., & Beltrán, M. (2019). Understanding and mitigating OpenID Connect threats. Computers & Security, 84, 1-16.
71	R38	EFFECT -Security	Li, W., Mitchell, C. J., & Chen, T. (2019). Oauthguard: Protecting user security and privacy with oauth 2.0 and openid connect. In Proceedings of the 5th ACM Workshop on Security Standardisation Research Workshop (pp. 35-44).
72	R39	EFFECT -Security	Mladenov, V., Mainka, C., & Schwenk, J. (2015). On the security of modern single sign-on protocols: Second-order vulnerabilities in openid connect. arXiv preprint arXiv:1508.04324.
73	R40	EFFECT	Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural patterns for microservices: a systematic mapping study. In CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018. SciTePress.
74	R41	EFFECT	Tighilt, R., Abdellatif, M., Moha, N., Mili, H., Boussaidi, G. E., Privat, J., & Guéhéneuc, Y. G. (2020). On the study of microservices antipatterns: A catalog proposal. In Proceedings of the European Conference on Pattern Languages of Programs 2020 (pp. 1-13).
75	R42	EFFECT	Behnel, S., Fiege, L., & Muhl, G. (2006). On quality-of-service and publish-subscribe. In 26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06) (pp. 20-20). IEEE.
76	R43	EFFECT	Cugola, G., Margara, A., & Migliavacca, M. (2009). Context-aware publish-subscribe: Model, implementation, and evaluation. In 2009 IEEE symposium on computers and communications (pp. 875-881). IEEE.
77	R44	EFFECT -Security	Costa, P., Migliavacca, M., Picco, G. P., & Cugola, G. (2004). Epidemic algorithms for reliable content-based publish-subscribe: An evaluation. In 24th International Conference on Distributed Computing Systems, 2004. Proceedings. (pp. 552-561). IEEE.
78	R45	EFFECT -Performance	Lazidis, A., Tsakos, K., & Petrakis, E. G. (2022). Publish–Subscribe approaches for the IoT and the cloud: Functional and performance evaluation of open-source systems. Internet of Things, 19, 100538.

79	R46	EFFECT -Performance	Oh, S., Kim, J. H., & Fox, G. (2010). Real-time performance analysis for publish/subscribe systems. Future Generation Computer Systems, 26(3), 318-323.
80	R47	EFFECT -Security	Wardana, A. A., & Perdana, R. S. (2018). Access control on internet of things based on publish/subscribe using authentication server and secure protocol. In 2018 10th International Conference on Information Technology and Electrical Engineering (ICITEE) (pp. 118-123). IEEE.