Published in Requirements Engineering Journal – Pre-Print – The Correct and Final Version

RAPID: A Knowledge-Based Assistant for Designing Web APIs

Mahsa H. Sadi & Eric Yu

Department of Computer Science
University of Toronto, Toronto, Canada
{mhsadi,eric} @cs.toronto.edu

Abstract

With the rise in initiatives such as software ecosystems and Internet of Things (IoT), developing web Application Programming Interfaces (web APIs) has become an increasingly common 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 no framework to specifically assist software developers in addressing these requirements in web APIs. In this paper, we introduce Rational API Designer (RAPID), an open-source assistant that advises on designing non-functional requirements in the architecture of web APIs. We have equipped RAPID with a broad range of expert knowledge about API design, systematically collected and extracted from the literature. The API design knowledge has been encoded as a set of 156 rules using the Non-Functional Requirements (NFR) multi-valued logic, a formal framework commonly used to describe non-functional and functional requirements of software systems. RAPID uses the encoded knowledge in a stepwise inference procedure to arrive from a given requirement, to a set of design alternatives, to a final recommendation for a given API design specification. Seven well-experienced software engineers have blindly evaluated the accuracy of RAPID's consultations over seven different cases of web API design and on providing design guidelines for thirty design questions. The results of the evaluation show that RAPID's recommendations meet acceptable standards of the majority of the evaluators 73.3% of the time. Moreover, analysis of the evaluators' comments suggests that more than one-third of the unacceptable ratings (33.8%) given to RAPID's answers are due to valid but incomplete design guidelines. We thus expect that the accuracy of the consultations will increase as RAPID's knowledge of API design is extended and refined.

Keywords

Non-Functional Requirements, Software Design and Architecture, Search-Based Software Engineering, Recommendation Systems, Chatbots, Knowledge-Based Question Answering, Automated Design Generation.

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 provide online programmatic endpoints for their back-end applications and services. One main concern in developing web APIs is that they expose back-end services and data towards clients. This exposure raises serious concerns about critical non-functional requirements, such as the security of back-end systems, the confidentiality and privacy of exchanged messages and data, and the performance of provided services [4, 5, 6]. 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) [7]. 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. These trade-offs become difficult to be deliberately assessed and are prone to be neglected as the number of design decisions increases.

The Account API Specification:

The Account API provides customer account data to third-party clients. The API only provides read access to the clients.

Two possible use cases of the Account API are as follows: (a) To aggregate financial metrics of a customer: For example, the client application aggregates and shows the information of all accounts of a customer (such as the balance of several bank accounts). (b) To recommend investment options to a customer or to provide money management guidelines.

Functional Requirements:

- Given a customer name and the Account ID, the Account API returns the account data.

Non-Functional Requirements:

- *Confidentiality and Privacy are Very Critical*: The Account API provides sensitive customer account data to the clients. Thus, confidentiality and privacy of the provided data are two critical requirements for the API. The Account API should be protected against unauthorized and unintended access as well as security attacks, such as identity theft attacks and eavesdropping attacks (i.e., unauthorized listening to the requests and responses of the API).
- *Latency is Critical:* The authorized clients should be able to access the account information in a timely manner.

The summary of the “Account API” Specification is provided in the following table:

System	Banking Platform
Name	Account API
Type of API	Public (Open) Web API
Expected Functionality	Providing the Account Data of a Given Customer
Important Requirements	(1) Confidentiality of the API is very critical. (2) Privacy of the API is very critical. (3) Latency of the API is critical.

Figure 1. A Simplified Instance of a Web API Requirements Specification

To clarify the problem, we use the example of developing an “Account API” for a banking platform. Consider the case of online and mobile banking. Information about customer accounts is going to be accessible over the Web via an “Account API” that the account management system provides. The “Account API” will be used by various clients including desktop, mobile and third-party applications and services. The clients can access customer accounts to read and display account information. Consider that a developer is provided with a requirements specification as shown in Figure 1. (We have simplified the specification to provide an overall sense of some important non-functional requirements that should be considered when developing a web API.) One design decision that the developer should make in developing the “Account API” is about securing access to the API. When designing a mechanism to secure access to the “Account API”, the developer should consider all the non-functional requirements important to the case (including privacy and confidentiality of the provided data as well as latency in responding to the client applications). She should also analyze the impact of candidate designs on these requirements before selecting one.

Currently, knowledge about addressing non-functional requirements in the design of web APIs is scattered among various heterogeneous online resources, including books, vendor white papers and weblogs written by practitioners involved in developing web APIs (e.g., [8, 9, 10, 11, 12]), or disparate bodies of design standards and scholarly articles written by researchers (e.g., [13, 14, 15, 16, 17, 18,19,20, 21, 22]). However, this dispersity impedes the on-demand use of the available design knowledge. Developers in need of the required information, would need to spend considerable time and use a web search engine such as Google or Google Scholar to search and find the related knowledge sources, sift through the retrieved resources based on some criteria, study the selected resources, extract different pieces of information from multiple resources and finally draw conclusions based on the performed research. Nevertheless, despite the importance of this information in design decision making, very little research efforts have been devoted to developing tools that aid software developers with obtaining the related information.

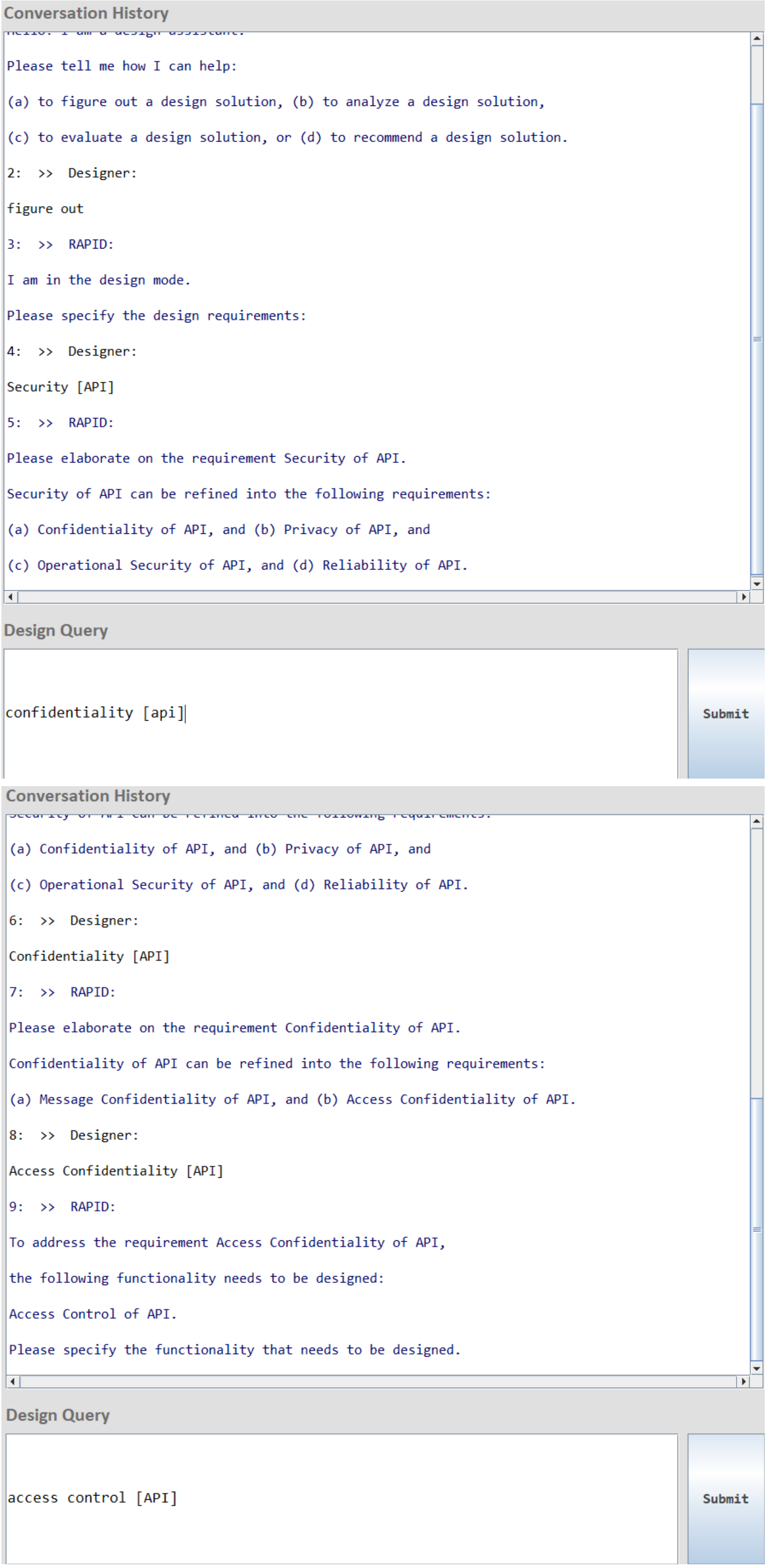
To address the above gap, in this paper, we introduce Rational API Designer (RAPID), a novel conversational assistant that aids software developers in addressing non-functional requirements in the design of web APIs. RAPID provides three types of consultations to the users: (a) introducing a set of techniques to design a non-functional or related functional requirement, (b) analyzing the trade-offs of a given design technique against non-functional requirements, and (c) recommending a final set of design alternatives for a given case. RAPID helps eliminate some of the time-consuming research steps that developers should perform on their own to figure out how to address non-functional requirements in the design of APIs.

In Figure 2 to Figure 4, we demonstrate how RAPID assists the developer to design a mechanism to secure access to the “Account API”. Figure 2 shows how RAPID helps the developer elaborate on the requirement “Security of API” and specify the exact intent and topic of the design query, i.e., “Access Confidentiality of API”. Figure 3 shows how RAPID helps the developer transform a non-functional requirement into a set of concrete design techniques and how it informs the developer of the trade-offs of a design technique. Figure 4 shows how RAPID recommends a final set of design techniques considering all the requirements important to the design of the “Account API” and their priority.

We have designed and developed RAPID as a rule-based knowledge-based system following five research steps:

1. *Design knowledge acquisition*: We have systematically extracted, structured and summarized the knowledge about addressing non-functional requirements in the design of web APIs from disparate online resources.
2. *Formal knowledge representation*: We have encoded the collected body of knowledge into a set of implication rules.
3. *Knowledge reasoning:* We have devised a stepwise inference procedure that transforms textual requirements specifications into functional designs and recommends design fragments using the encoded knowledge.
4. *Knowledge base implementation*: We have implemented the working prototype of a knowledge-based system that uses the encoded knowledge and follows the step-wise inference procedure to provide design guidelines.
5. *Knowledge base verification and validation*: We have asked seven experienced software engineers to blindly evaluate the accuracy of RAPID’s recommendations in a web API design exam containing thirty questions.

In the rest of this paper, we delineate the above steps: In section 2, we describe the procedure for collecting and organizing design knowledge. In section 3, we explain how to formalize the organized design knowledge. In section 4, we present the step-wise inference procedure for using the encoded knowledge. In section 5, we explain the design and implementation of RAPID. In section 6, we present the evaluation methodology. In section 7, we review the related research. Finally, in section 8, we discuss the reported research and outline avenues for future research.



Hello: I am a design assistant.

Please tell me how I can help:

(a) to figure out a design solution, (b) to analyze a design solution,

(c) to evaluate a design solution, or (d) to recommend a design solution.

2: >> Designer:

figure out

3: >> RAPID:

I am in the design mode.

Please specify the design requirements:

4: >> Designer:

Security [API]

5: >> RAPID:

Please elaborate on the requirement Security of API.

Security of API can be refined into the following requirements:

(a) Confidentiality of API, and (b) Privacy of API, and

(c) Operational Security of API, and (d) Reliability of API.

confidentiality [api]

Security of API can be refined into the following requirements:

(a) Confidentiality of API, and (b) Privacy of API, and

(c) Operational Security of API, and (d) Reliability of API.

6: >> Designer:

Confidentiality [API]

7: >> RAPID:

Please elaborate on the requirement Confidentiality of API.

Confidentiality of API can be refined into the following requirements:

(a) Message Confidentiality of API, and (b) Access Confidentiality of API.

8: >> Designer:

Access Confidentiality [API]

9: >> RAPID:

To address the requirement Access Confidentiality of API,

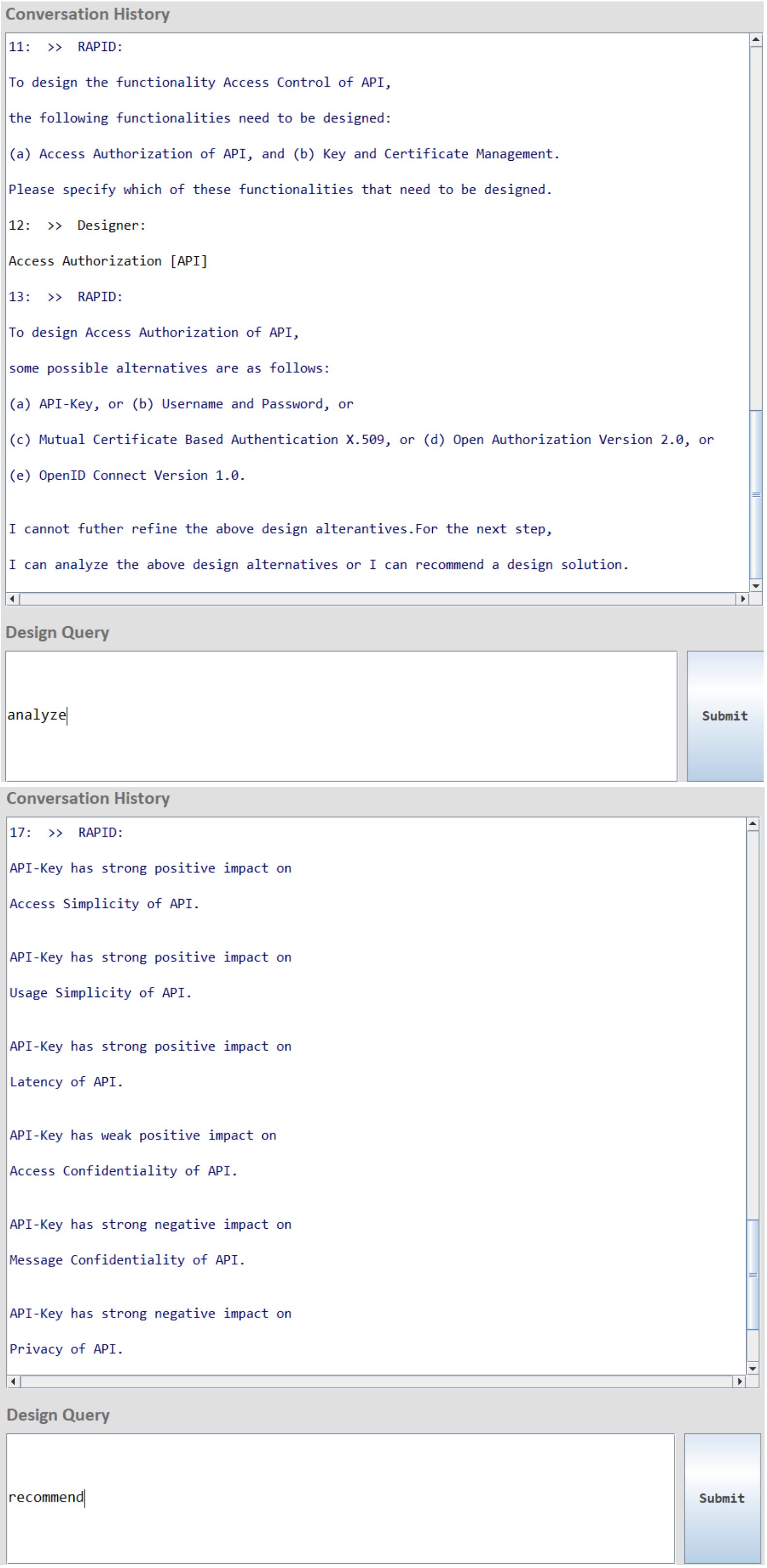
the following functionality needs to be designed:

Access Control of API.

Please specify the functionality that needs to be designed.

access control [API]

Figure 2. RAPID Helps the User Elaborate on a Given Requirement and Identify the Exact Intent of the Design Query



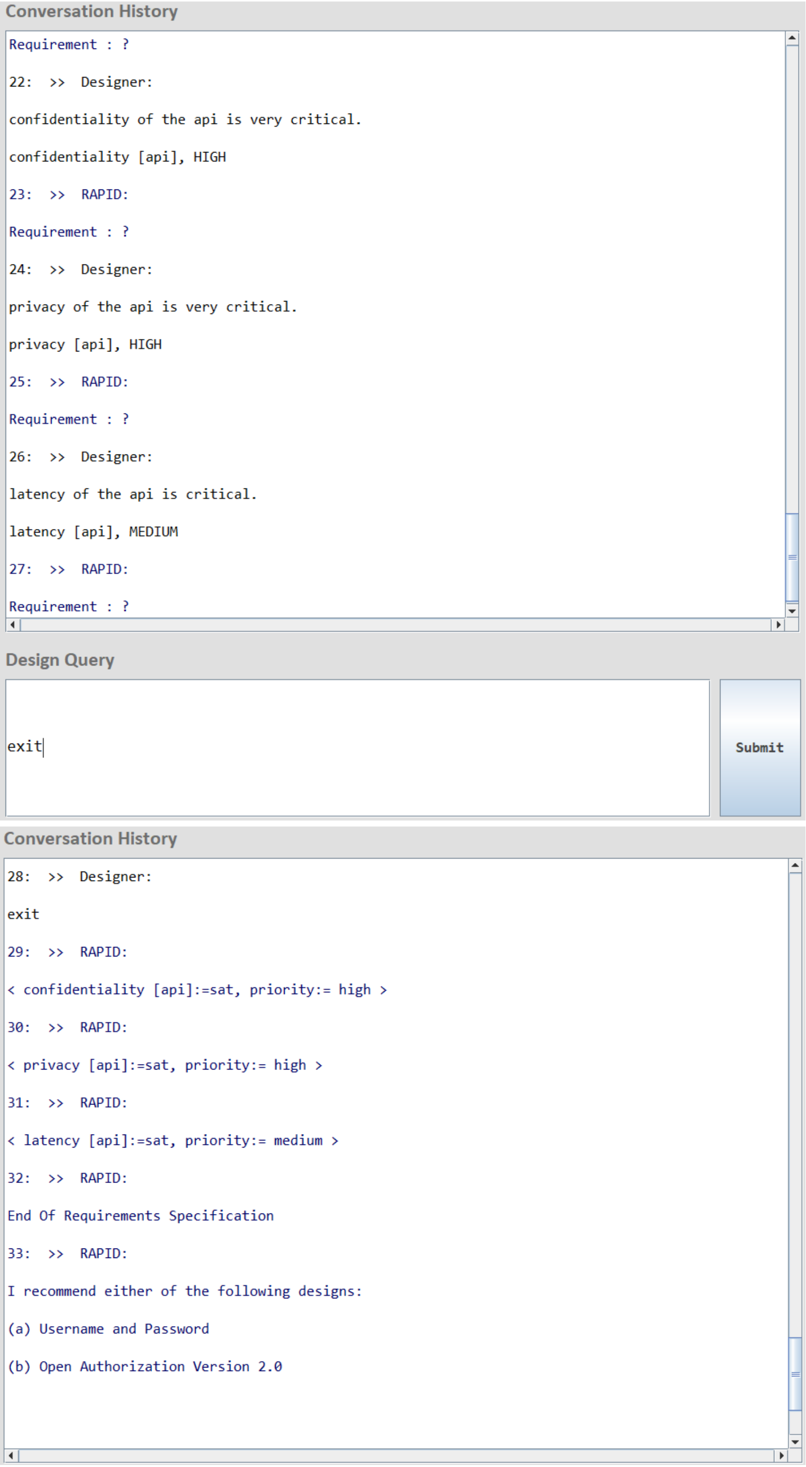
```
11:  >>  RAPID:

To design the functionality Access Control of API,

the following functionalities need to be designed:

(a) Access Authorization of API, and (b) Key and Certificate Management.

Please specify which of these functionalities that need to be designed.

12:  >>  Designer:

Access Authorization [API]

13:  >>  RAPID:

To design Access Authorization of API,

some possible alternatives are as follows:

(a) API-Key, or (b) Username and Password, or

(c) Mutual Certificate Based Authentication X.509, or (d) Open Authorization Version 2.0, or

(e) OpenID Connect Version 1.0.


I cannot futher refine the above design alterantives.For the next step,

I can analyze the above design alternatives or I can recommend a design solution.
```

analyze

```
17:  >>  RAPID:

API-Key has strong positive impact on

Access Simplicity of API.


API-Key has strong positive impact on

Usage Simplicity of API.


API-Key has strong positive impact on

Latency of API.


API-Key has weak positive impact on

Access Confidentiality of API.


API-Key has strong negative impact on

Message Confidentiality of API.


API-Key has strong negative impact on

Privacy of API.
```

recommend

Figure 3. (Top) RAPID Helps the User Transform the Given Requirement into Design Techniques, (Bottom) RAPID Informs User about the Trade-Offs of a Design Technique



```
Requirement : ?

22:  >>  Designer:

confidentiality of the api is very critical.

confidentiality [api], HIGH

23:  >>  RAPID:

Requirement : ?

24:  >>  Designer:

privacy of the api is very critical.

privacy [api], HIGH

25:  >>  RAPID:

Requirement : ?

26:  >>  Designer:

latency of the api is critical.

latency [api], MEDIUM

27:  >>  RAPID:

Requirement : ?
```

```
exit
```

```
28:  >>  Designer:

exit

29:  >>  RAPID:

< confidentiality [api]:=sat, priority:= high >

30:  >>  RAPID:

< privacy [api]:=sat, priority:= high >

31:  >>  RAPID:

< latency [api]:=sat, priority:= medium >

32:  >>  RAPID:

End Of Requirements Specification

33:  >>  RAPID:

I recommend either of the following designs:

(a) Username and Password

(b) Open Authorization Version 2.0
```

Figure 4. (Top) RAPID Receives the Requirements Important to the Design Case and their Priority, (Bottom) RAPID Recommends a Final Set of Design Techniques considering all the Requirements of the Design Case

2 Collecting and Organizing Design Knowledge

To collect and organize the design knowledge, we have performed three steps: (1) We have conducted a systematic [23] and evidence-based review [24] of the literature to collect the related knowledge sources. (2) We have extracted the relevant pieces of design knowledge from the collected resources and aggregated and organized them into three categories: (a) common non-functional requirements that should be considered in the design of APIs, (b) techniques and mechanisms for addressing these non-functional requirements in APIs, and (c) the trade-offs of some of the design techniques against the identified non-functional requirements. (3) We have then visualized and summarized the structured design knowledge using knowledge graphs. The outcome of this step is a body of structured knowledge about designing non-functional requirements in the architecture of APIs, mainly formed based on expert opinion. In the following, we explain the details of each step.

2.1 Collecting Design Knowledge

The research questions driving the collection of the knowledge resources are 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?

The knowledge addressing the above research questions appears in two forms:

- *Expert Opinion*: Expert opinion is the written experience and opinion of experts (i.e., practitioners) about the design and development of APIs. This kind of resources provides guidelines, patterns, best practices, and mechanisms for designing web APIs. Expert opinion can be found in two types of resources: (a) Books written by experts about the design and architecture of APIs and API management platforms, (b) Informal online white papers, weblogs and written and video tutorials about designing APIs.
- *Scholarly Articles and Scientific Experiments*: Scholarly articles are written by scholars and researchers suggesting approaches and techniques for designing APIs. This kind of literature follows a scientific approach to suggest design techniques and the results are evaluated using a comparative or experimental approach.

The knowledge sources that address the above research questions can be categorized into the following topics: (a) API design and management, (b) APIs and software ecosystems, and (c) Web-services, service-oriented architectures, micro-services, and cloud design.

To search the online knowledge resources, we have used two web search engines, namely Google Scholar, and Google. We have used Google Scholar to find books and scholarly articles. We have used Google search engine to find weblogs, tutorials, and white papers. The search process has been performed in several rounds (similar to a backward snow-balling process [25] but also including resources that are not scholarly and do not have references). In the first round, a set of resources (mainly books written by experts) 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.

To search for the related resources, we have used the combination of the following query keywords: “API”, “Application Programming Interface”, “Design”, “Development”, and “Management”. To search for non-functional requirements, we have used the combination of the following keywords: “Requirement”, “Quality Attribute”, “Quality of Service”, “Non-Functional”, and “API” and “Application Programming Interface”. To search for the available mechanisms, techniques, patterns, and guidelines, we have used the combination of the following keywords: “Design”, “Pattern”, “Mechanism”, “Technique”, “API”, “Application Programming Interface”, and “API Management”. To search for evidence about the impact of the available techniques, mechanisms, and patterns on the identified non-functional requirements, we have used the combination of the following keywords: "Analysis", "Evaluation", "Strength", "Weakness" and "Trade-off", and each of the identified non-functional requirements and the identified design techniques or patterns.

2.2 Extracting, Classifying, and Aggregating Design Knowledge

To extract and classify the information addressing the above research questions, we have used specific templates.

The template used for extracting and analyzing information about the non-functional requirements includes the following items: (a) the name of the non-functional requirement, (b) the definition of the non-functional requirement, (c) the qualitative or quantitative metrics and measures to evaluate non-functional requirements (if available in the collected resources), and (d) the knowledge sources containing the information.

To categorize the identified non-functional requirements, we have performed four steps: (1) We have used ISO / IEC 25011 standard [26] for system and software quality requirements and evaluation to check the definition of non-functional requirements and to classify similar quality requirements. In some cases, a quality requirement (i.e., non-functional requirement) as defined by IEC was 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) 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. Two requirements are considered as different if they differ in definition, or different metrics and criteria and measures are used to evaluate them. (3) We have grouped similar non-functional requirements that refer to the same type of non-functional requirement. (4) 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 used to assess them. An example of classified information about an API non-functional requirement is shown in Table 1.

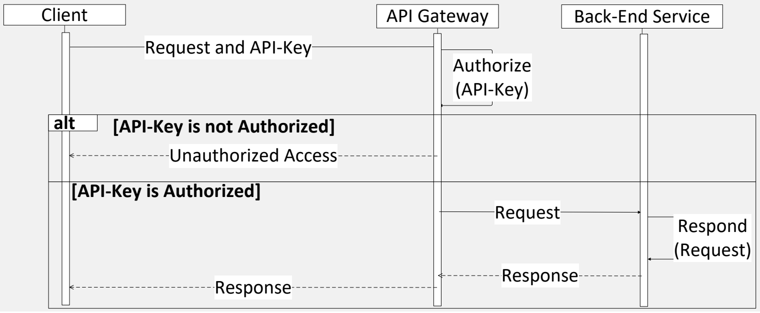
Table 1. An Instance of Extracted and Classified Information about a Non-Functional Requirement

Non-Functional Requirement	- API Usability or API Developer Experience (DevX)
Definition	- Usability of an API is the degree of ease with which developers can use the API in their code to achieve their development goals.
Requirement Sub-Types	- *Learnability* (*Understandability)*: How easy it is to learn and understand the API. Learnability include memorability quality attribute. - *Memorability*: How easily the API calls can be remembered. - *Efficiency*: How efficiently the API can be used for specific tasks. **-** *Usage Simplicity:* How simple it is to use the API. **-** *Consistency***:** How well an API matches the developers' mental model.
Evaluation Metrics or Measures	- Learning time of the API users
Knowledge Source	McLellan et al, 1998 [27] , Stylos, & Myers 2007 [19], Robillard, 2009 [28], Robillard, & Deline, 2011 [29], Piccioni et al., 2013 [30], Myers & Stylos, 2016 [6].

The template used for extracting information about the mechanisms for addressing non-functional requirements in APIs contains the following items (The used structure is close to the template of design patterns introduced in [31]): (a) the name of the design mechanism or technique, (b) the objective of the design, (c) the description of design, (d) the model of the design (if available), (e) the applicability condition of the mechanism, explaining when the design can be used (if available), and (f) the knowledge sources containing the information.

To categorize the design mechanisms, we have performed four steps: (1) We have abstracted away the details of that depend on development methodology (such as micro-service or service-oriented approach) or the choice of implementation (such as REST APIs or SOAP APIs). (2) We have merged the design mechanisms that implement the same approach but are referred to with different terms. (3) We have classified the mechanisms based on their design objectives; i.e., the functionality that the mechanisms provide or the non-functional requirement that they address. (4) We have related the design objectives of the mechanisms to the non-functional requirement that they address. An example of classified information about a design technique is shown in Table 2.

Table 2. An Instance of Extracted and Classified Information about a Design Technique

Design Technique	- API-Key (Client-ID, App-Key, App-ID, Consumer-ID)
Design Objective	- To identify and authorize the client of an API
Category	- API security mechanism, API access control mechanism, API authentication and authorization mechanism
Description	- API key is a unique key issued by the API provider when a developer registers her application in order to use an API. An API Key identifies the application using the API. API Key is a long alpha-numeric string that is opaque and without any signature and encryption.
Sequence Diagram	 Client API Gateway Back-End Service Request and API-Key Authorize (API-Key) alt [API-Key is not Authorized] Unauthorized Access [API-Key is Authorized] Request Respond (Request) Response Response
Applicability Condition	- Used to identify and authorize the application or service requesting the service. - API-key can be used with other API security mechanisms for better security.
Knowledge Source	De, 2017 [9]

The template used for extracting and analyzing evidence about the impact of the identified design mechanism on non-functional requirements includes the following items: (a) the name of the mechanism or 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.

We have categorized the identified trade-offs based on 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 an empirical evidence extracted from the collected literature. 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 when the effect is neither weak, nor strong. "To some extent" impact has been used when the extent of the impact could not be identified, or when the impact has been neither weak, nor strong. "Strong", and "Weak" labels are used when there is evidence that the effect of a technique on a non-functional requirement is strong or weak. The type of evidence and justification for the identified effects are categorized as anecdotal evidence, expert opinion, metric-based (if some criterion or measure is used to justify the type and extent of the effect), or scientific experiments. The type of evidence is identified according to the type of argument used to justify the type and extent of an effect. An example of extracted information about the trade-offs of a design technique is shown in Table 3.

Table 3. An Instance of Extracted and Classified Information about the Trade-Offs of a Design Technique

Design Technique	API-Key			
Non-Functional Requirement	**Effect Type**	**Extent of Effect**	**Justification and Reasons for Evaluation**	**Evidence**
Access Simplicity of API	+	Strong	- The potential clients can easily access the API and the back-end system via registering with the API provider and obtaining an API key. The only barrier and security check that is performed to access an API is providing the key which can be easily obtained.	- Expert opinion
Usage Simplicity of API	+	Strong	- An API can be simply used by presenting a key to the API. There are low security barriers in order to use an API.	- Expert Opinion
Access Confidentiality of API	+	Weak	- API-Key provides low level of confidentiality for accessing to an API. API key is the minimum verification mechanism that can be added to authenticate a client. The client (the provider of the key) is not authenticated.	- Expert Opinion
Message Confidentiality of API	-	Strong	- API-Key is not encrypted or signed. This exposes the API-key to eaves-dropping attacks.	- Expert Opinion
Privacy of API	-	Strong	- The API-key does not support any mechanism for ensuring the privacy of the services and data that an API provides. The client (the provider of the key) is not authenticated.	- Expert Opinion
Latency of API	+	Strong	- Low overhead is added to the communications between a client and an API provider in order to allow access to an API.	-Expert Opinion - Metric Based: Number of additional interactions to access an API
Related Knowledge Source			De, 2017 [9]	

2.3 Summarizing and Visualizing Design Knowledge

We have summarized and visualized the structured knowledge using knowledge graphs [32]. The knowledge graphs are directed and typed, and illustrate the relationships between the identified non-functional requirements and design techniques. The nodes in the knowledge graphs represent three entities: (a) non-functional requirements, (b) design objectives, and (c) design mechanisms. These entities are identified based on the "Non-Functional Requirement" and "Requirements Sub-Types" fields of the template used to extract information about non-functional requirements, and the "Design Technique", "Design Objective", and "Category" fields of the template used to extract information about design mechanisms. The edges are directed and typed and represent directed relationships between entities. The edges have one of the following types: "IS-A", "REALIZES", "Strong+", "Some+", "Weak+", "Strong–", "Some–", "Weak–". "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 other labels show the type and extent of the effect of a design mechanism on the related non-functional requirements and are developed based on the "Effect Type" and "Extent of Effect" fields in the template used to extract information about trade-offs. The constructed knowledge graphs meet two constraints: (a) The nodes representing non-functional requirements or design objectives may have one to many incoming edges, but only one outgoing edge. (b) There is no cycle in the graphs.

Four instances of knowledge graphs are shown in Figure 5 to Figure 9. Figure 5 is a part of a knowledge graph showing the decomposition relationship between “API Usability” non-functional requirement and its sub-types, and is developed based on Table 1. Figure 6 is a part of a knowledge graph showing alternative designs for “Authorizing Access to an API”, and is developed based on Table 2 and some other tables with the same “Design Objective”, and “Category” fields as in Table 2. Figure 7 is a part of a knowledge graph relating a non-functional requirement to the design techniques that address the requirement and is developed by connecting knowledge graphs of non-functional requirements (such as shown in Figure 5) to knowledge graphs of design techniques (such as shown in Figure 6). Figure 9 is a part of a knowledge graph showing the trade-offs of a design mechanism and is developed based on Table 3. Appendix 1 contains the knowledge graphs summarizing the collected and organized body of web API design.

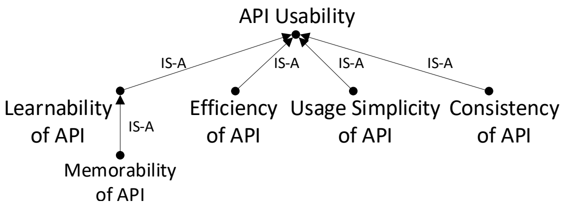


Figure 5. A Part of Knowledge Graph Related to a Non-Functional Requirement

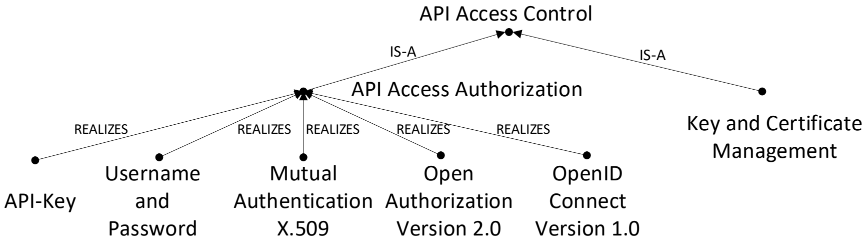


Figure 6. A Part of a Knowledge Graph Related to Design Techniques

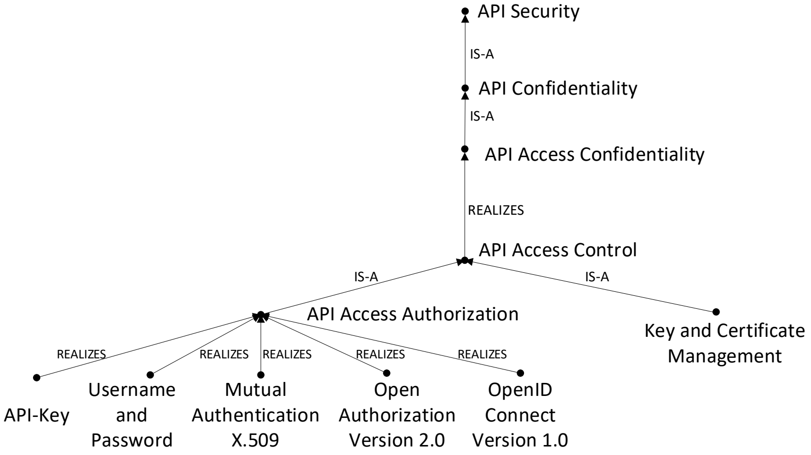


Figure 7. A Part of a Knowledge Graph Relating Some Design Techniques to a Non-Functional Requirement

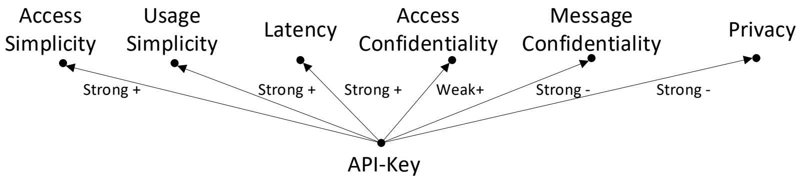


Figure 8. A Part of a Knowledge Graph Related to Trade-Offs of a Design Mechanism

3 Formalizing and Encoding Design Knowledge

To encode the design knowledge, we have used the NFR multi-valued logic [33] (see Appendix 2) a formal framework commonly used to describe both non-functional and functional requirements of software systems. The multi-valued nature of the NFR logic allows to describe both non-functional and functional aspects of the design.

The input to the formalization step is the design knowledge graphs constructed in the previous step. The output is a set of rule graphs encoding the same knowledge. Each rule in a rule graph is an implication rule with the following general form:

$(Term_1, \ldots, Term_m) \xrightarrow{Rule\ Type} Term_n : Rule\ Category$

where $m \geq 1$,

$Term_i$ is a string of the form "Type [Topic]",

Rule Type ∈ {Break, SomeMinus, Hurt, Unknown, Help, SomePlus, Make, AND, OR, XOR[1]}, and

Rule Category ∈ {NF-REF, NF-OP, F-REF, F-OP, COR},

Satisfaction ($Term_i$) ∈ {Denied, Partially Denied, Unknown, Conflict, Partially Satisficed, Satisficed}.

A rule describes a relation between two sets of terms, referred to as precedent and consequent. A precedent is composed of one or more terms, while a consequent has exactly one term. Precedent and consequent are related to each other via a rule type. A rule type can adopt values from the set {Break, SomeMinus, Hurt, Unknown, Help, SomePlus, Make, AND, OR, XOR}. A rule has also a rule category adopting a value from the set {NF-REF, NF-OP, F-REF, F-OP, COR}. Each term in the precedent or consequent of a rule can adopt a satisfaction value from the set {Denied, Partially Denied, Undetermined, Unknown, Conflict, Partially Satisficed, or Satisficed}.

The relationship between two or more rules is represented as a rule graph. A rule graph is directed and acyclic graph. The nodes in a rule graph represent precedents and consequents and the edges represent the "rule type" and "rule category" connecting the nodes. A rule graph has one constraint: There are no two distinct rules that have the same consequent unless the category of at least one of these rules is *COR*. This constraint helps avoid conflict in selecting, retrieving, and evaluating rules in the inference procedure as described in the next step. The rules encoding the collected body of web API Design knowledge are listed in Appendix 3.

To transform the design knowledge graphs into rule graphs, we have performed two common steps: (1) Turning each node in the knowledge graph into a term in the general form of "Type [Topic]". (2) Turning each directed edge into an implication rule (→), a "Rule Type" and a "Rule Category". The "Rule Type" and "Rule Category" are identified based on the category of entities in the knowledge graphs, as described in the rest of this section.

3.1 Encoding the Knowledge Related to Non-Functional Requirements

Knowledge graphs related to non-functional requirements are encoded into rules with the following general form:

$$(Term_1, \ldots, Term_m) \xrightarrow{Rule\ Type} Term_n : \mathrm{NF-REF} \quad , Rule\ Type \in \{\mathrm{Help, AND}\}$$

"IS-A" edges connecting two or more entities of type non-functional requirement are encoded into "Non-Functional Requirement Refinement" (NF−REF) rule category. The NF−REF rule category identifies that the terms in the precedent of the rule are non-functional requirements that refine and decompose the non-functional requirement term in the consequent of the rule. One-to-one and many-to-one edges between nodes are encoded as "Help" and "AND" rule

---

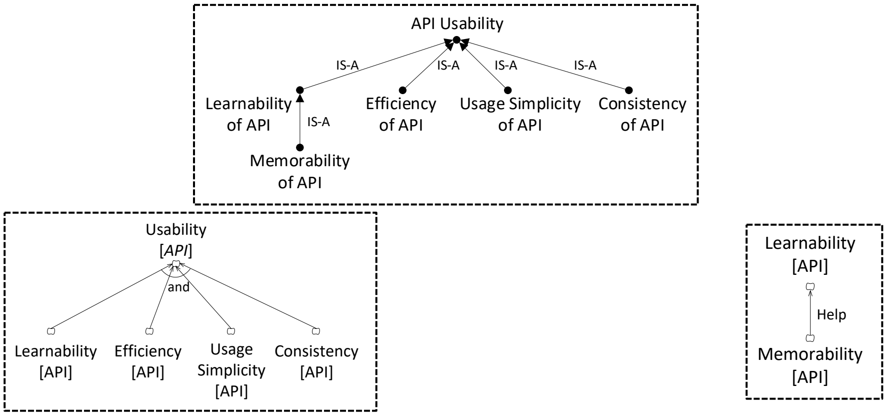
[1] In the original NFR framework, XOR relationship between a set of variables is not explicitly defined. We define an XOR contribution of the form $(G_i,G_j) \xrightarrow{xor} G_k$ between three variables of G_i, G_j, and G_k as following: G_k is satisficeable if one and only one of the two variables G_i and G_j is satisficed and the other variable is denied assuming that the interdependency itself is satisficed (i.e., the XOR contribution holds true). XOR contribution is an associative relation.

types respectively. If a non-functional requirement is related to a single non-functional requirement, the edge is encoded as "Help", conveying that the fulfillment of the requirement appearing in precedent provides partial positive support for the fulfillment of the requirement in consequent. If more than one non-functional requirement are related to a non-functional requirement, the edges between the source nodes and the destination node are encoded as "AND", implying that the un-fulfillment of each of the requirements in the precedent affects the fulfillment of the requirement in the consequent (see Table 4). It is possible that some of the terms in the precedent of an "AND" rule are not relevant to a specific case. In this circumstance, the effect of unrelated terms will be neutralized in the inference procedure described in the next step.

Table 4. Rule Representation of "IS-A" Relationships between Non-Functional Requirements

Rule	**Simplified English Translation**
$(Term_i) \xrightarrow{Help} Term_j\text{: NF} - \text{REF}$	"The requirement $Term_i$ refines the requirement $Term_j$."
$(Term_1, \ldots, Term_m) \xrightarrow{and} Term_n : \text{NF} - \text{REF}$	"The requirements $Term_j, \ldots, Term_m$ refine the requirement $Term_n$.

An example of formalizing a knowledge graph related to a non-functional requirement is shown in Figure 9.



$$\text{(Learnability [API], Efficiency [API], Usage Simplicity [API], Consistency [API])} \xrightarrow{\text{and}} \text{Usability [API] : NF-REF}$$

$$\text{(Memorability [API])} \xrightarrow{\text{Help}} \text{Learnability [API] : NF-REF}$$

Figure 9. An Instance of Formalizing a Part of a Knowledge Graph Related to Non-Functional Requirements – (Top) Knowledge Graph, (Bottom) Related Rules

3.2 Encoding the Knowledge Related to Design Techniques

In the knowledge graphs related to design mechanisms, there are two types of edges: (a) "IS-A" edges relate two or more entities of type design objectives, denoting that the source nodes decompose the destination node. (b) "REALIZES" edges relate one or more entities of type design mechanism to an entity of type design objective, conveying that the source node(s) are alternative mechanisms to implement the destination node. These two edge types are respectively transformed into two categories of rules: (a) "Functional Requirements Refinement" (F−REF), and (b) "Functional Requirements Operationalization" (F−OP) as described in the following:

3.2.1 Functional Requirement Refinement (F-REF) Rules

Functional requirement refinement rules encode the knowledge about how to decompose a high-level design objective into more fine-grained ones. "F−REF" rule category identifies that the terms in the precedent of the rule are a set of design objectives that refine and decompose the design objective term in the consequent of the rule. The general form of functional refinement rules is as follows:

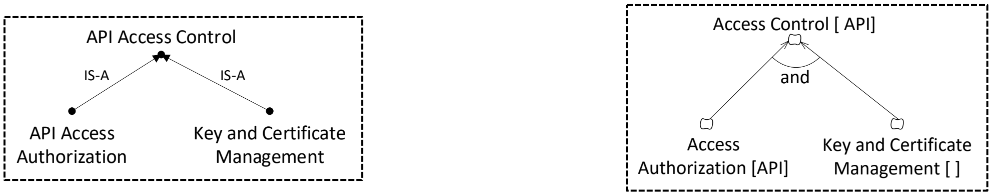
$$(Term_1, \ldots, Term_m) \xrightarrow{Rule\ Type} Term_n : F{-}REF \qquad , Rule\ Type \in \{\text{Make, OR, AND}\}$$

One-to-one and one-to-many edges in knowledge graphs are encoded into “Make”, “AND” and “OR” rule types respectively: If one design objective is related to another design objective, the relationship between the two is of type “MAKE”, implying that the fulfillment of the design objective in the precedent leads to the fulfillment of the design objective in the consequent. If two or more design objectives are related to a design objective, the relationship between the two groups can be of type “AND”, or “OR”. “AND” is when all the objectives in the precedent are required to address the objective in the consequent. “OR” is used when any sub-set of the objectives in the precedent is enough and can be used to address the objective in the consequent (see Table 5).

Table 5. Rule Representation of “IS-A” relationships between Design Objectives

Rule	**Simplified English Translation**
$(Term_i) \xrightarrow{Make} Term_j\text{: F} - \text{REF}$	“To design $Term_j$, $Term_i$ should be designed .”
$(Term_1, \ldots, Term_m) \xrightarrow{\text{and / } or} Term_n\text{: F} - \text{REF}$	“To design $Term_n$, the functionalities $Term_1$,... and/or $Term_m$ should be designed.”

An example of a formalized knowledge graph related to a functional refinement is shown in Figure 10.


API Access Control
IS-A
IS-A
API Access Authorization
Key and Certificate Management
Access Control [API]
and
Access Authorization [API]
Key and Certificate Management [ ]

$$(\text{Access Authorization [API], Key and Certificate Management [ ]}) \xrightarrow{and} \text{Access Control [API] : F-REF}$$

Figure 10. An Instance of Formalizing a Part of Knowledge Graph Related to Decomposing Design Techniques – (Left) Knowledge Graph, (Right) Related Rules

3.2.2 Functional Requirement Operationalization (F-OP) Rules

Functional requirement operationalization rules encode the knowledge about how to implement a design objective. “$F{-}OP$” rule category identifies that the term in the precedent of the rule is a set of design mechanisms that implement the design objective in the consequent of the rule. The general form of these rules is as follows:

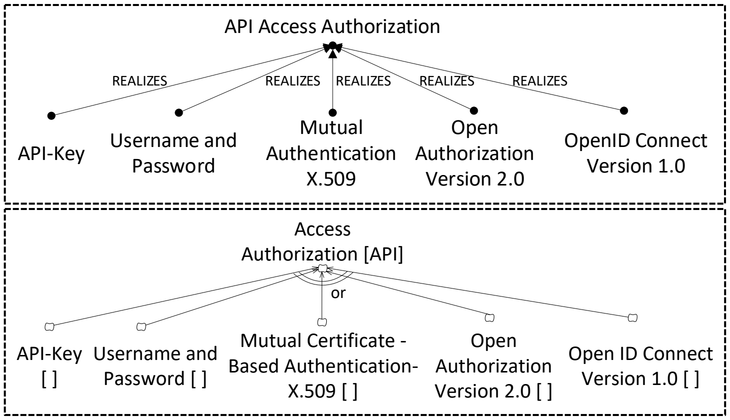
$$(Term_1, \ldots, Term_m) \xrightarrow{Rule\ Type} Term_n\text{: } F{-}OP \qquad , Rule\ Type \in \{\text{Make, OR, XOR}\}$$

One-to-one and one-to-many edges in knowledge graphs related to design techniques are encoded into “Help”, “OR” and “XOR” rule types respectively: If one design mechanism is related to a design objective in the knowledge graph, the relationship between the two is of type “MAKE”, implying that the fulfillment of the precedent design mechanism provides sufficient positive support for the fulfillment of the objective in the consequent. If one or more design mechanisms are related to a design objective, the relationship between the two groups can be of type “OR”, or “XOR”. “XOR” is used when the mechanisms are alternative to each other and can be used instead of each other. OR is used when any sub-set of the design mechanisms can be used to fulfill the design objective (see Table 6).

Table 6. Rule Representation of “REALIZES” Relationships between One or More Design Techniques and a Design Objective

Rule	**Simplified English Translation**
$(Term_i) \xrightarrow{Make} Term_j\text{: F} - \text{OP}$	“$Term_i$ is one alternative to design $Term_j$.”
$(Term_1, \ldots, Term_m) \xrightarrow{\text{or / xor}} Term_n\text{: F} - \text{OP}$	“$Term_1$, ..., $Term_m$ are some possible alternatives to design $Term_n$.

An example of a formalized knowledge graph related to a functional operationalization is shown in Figure 11.


API Access Authorization
REALIZES
REALIZES
REALIZES
REALIZES
REALIZES
API-Key
Username and Password
Mutual Authentication X.509
Open Authorization Version 2.0
OpenID Connect Version 1.0
Access Authorization [API]
or
API-Key [ ]
Username and Password [ ]
Mutual Certificate - Based Authentication- X.509 [ ]
Open Authorization Version 2.0 [ ]
Open ID Connect Version 1.0 [ ]

$$\text{(API-Key [ ], Username and Password [ ], Mutual Certificate-Based Authentication X.509 [ ], Open Authorization Version 2.0 [ ], Open-ID Connect Version 1.0 [ ])} \xrightarrow{or} \text{Access Authorization [API] : F-OP}$$

Figure 11. An Instance of Formalizing a Part of a Knowledge Graph Related to Design Techniques – (Top) Knowledge Graph, (Bottom) Related Rule Graph and Rule

3.3 Encoding the Relationships between Non-Functional Requirements and Design Techniques

Two groups of edges in the knowledge graphs relate design techniques to non-functional requirements: (a) “REALIZES” relationships and (b) Trade-offs. These two groups of edges are encoded as to categories of rules: (a) “Non-Functional Requirement Operationalization” (NF-OP) rules and “Correlation” (COR) rules.

3.3.1 Non-Functional Requirement Operationalization (NF-OP) Rules

“REALIZES” relationships in the knowledge graphs relating non-functional requirements to design objectives are encoded as “Non-Functional Operationalization” ($NF-OP$) rule category. “$NF-OP$” rule category identifies that the terms in the precedent of the rule are design objectives that realize the non-functional requirement term in the consequent of the rule. The general form of non-functional operationalization rules is as follows:

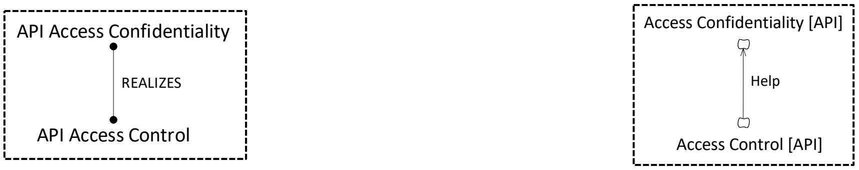
$$(Term_1, \ldots, Term_m) \xrightarrow{Rule\ Type} Term_n : NF-OP \qquad , Rule\ Type \in \{\text{Help, OR}\}$$

One-to-one and one-to-many “Realizes” relationships in knowledge graphs are encoded as “Help” and “OR” rule types respectively: If one design objective is related to a non-functional requirement, the relationship between the two is of type “Help”, implying that the fulfillment of the design objective provides partial positive support for the fulfillment of the non-functional requirement. If more than one design objectives are related to a non-functional requirement, the relationship between the two is of type “OR”, implying that the fulfillment of any subset of the objectives provides positive support for the fulfillment of the non-functional requirement (see Table 7).

Table 7. Rule Representation of “REALIZES” Relationships between Non-Functional Requirements and Design Objectives

Rule	**Simplified English Translation**
$(Term_i) \xrightarrow{Help} Term_j : \text{NF} - \text{OP}$	“To address the requirement $Term_j$, $Term_i$ needs to be designed .”
$(Term_1, \ldots, Term_m) \xrightarrow{or} Term_n : \text{NF} - \text{OP}$	“To address requirement $Term_n$, the functionalities $Term_1$, …, $Term_m$ need to be designed.”

An example of formalizing a part of a knowledge graph relating a design objective to a non-functional requirement is shown in Figure 12.



$$(\text{Access Control [API]}) \xrightarrow{Help} \text{Access Confidentiality [API]: NF-OP}$$

Figure 12. An Instance of Formalizing a Part of Knowledge Graph Relating a Non-Functional Requirement to a Design Technique – (Left) A Part of a Knowledge Graph, (Right) Related Part of the Rule Graph

3.3.2 Correlation (COR) Rules

The edges representing trade-offs of design mechanisms are encoded into correlation rules with the following form:

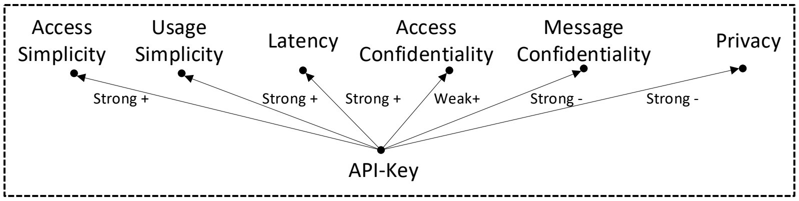
$$(Term_{\,i}) \xrightarrow{Rule\ Type} Term_{\,j} : COR \qquad , Rule\ Type \in \{\text{Break, SomeMinus, Hurt, Help, SomePlus, Make}\}$$

“COR” rule category identifies that the term identified in the precedent of a rule is a design mechanism which has a positive or negative impact on the non-functional requirement term in the consequent of the rule. Unlike other rules, there can be several correlation rules with the same consequent. Moreover, correlation rules have only one precedent and one consequent. Table 8 shows the mapping between the edge type in the knowledge graphs and the rule type.

Table 8. Mapping between Edge Type in the Knowledge Graphs and Rule Type in the Rule Graphs

Edge Type in the Knowledge Graph	Related Rule	Translation in English
Strong+	$(Term_{\,i}) \xrightarrow{Make} Term_{\,j} : COR$	“$Term_{\,i}$ has strong positive impact on $Term_{\,j}$.”
Some +	$(Term_{\,i}) \xrightarrow{Some+} Term_{\,j} : COR$	“$Term_{\,i}$ has some positive impact on $Term_{\,j}$.”
Weak +	$(Term_{\,i}) \xrightarrow{Help} Term_{\,j} : COR$	“$Term_{\,i}$ has weak positive impact on $Term_{\,j}$.”
Strong –	$(Term_{\,i}) \xrightarrow{Break} Term_{\,j} : COR$	“$Term_{\,i}$ has strong negative impact on $Term_{\,j}$.”
Some –	$(Term_{\,i}) \xrightarrow{Some-} Term_{\,j} : COR$	“$Term_{\,i}$ has some negative impact on $Term_{\,j}$.”
Weak –	$(Term_{\,i}) \xrightarrow{Hurt} Term_{\,j} : COR$	“$Term_{\,i}$ has weak negative impact on $Term_{\,j}$.”

An example of a formalized table related to the trade-offs a design technique is shown in Figure 13.



$$(\text{API-Key [ ]}) \xrightarrow{Make} \text{Access Simplicity [API] : COR}$$
$$(\text{API-Key [ ]}) \xrightarrow{Make} \text{Usage Simplicity [API] : COR}$$
$$(\text{API-Key [ ]}) \xrightarrow{Make} \text{Latency [API] : COR}$$
$$(\text{API-Key [ ]}) \xrightarrow{Help} \text{Access Confidentiality [API] : COR}$$
$$(\text{API-Key [ ]}) \xrightarrow{Break} \text{Message Confidentiality [API] : COR}$$
$$(\text{API-Key [ ]}) \xrightarrow{Break} \text{Privacy [API] : COR}$$

Figure 13. An Instance of Formalizing a Part of a Knowledge Graph Related to Trade-Offs – (Top) A Knowledge Graph, (Bottom) The Related Rules

4 The Step-Wise Inference Procedure

To use the encoded knowledge, we have devised a step-wise inference procedure. The input to the inference procedure is a set of terms in the form of “Type [Topic]” specifying the non-functional or related functional requirements of a design case. The final output is a set of terms describing concrete design techniques that address the input requirements. The inference procedure transforms a textual requirement specification into concrete design techniques using the rules developed in the previous step, and has three components: (a) a step-wise refinement procedure, (b) a step-wise analysis and evaluation procedure, and (c) a selection procedure. In the following, we explain each component.

4.1 The Step-Wise Refinement Procedure

The first component of inference is a step-wise refinement procedure that allows to reach from a specified non-functional or functional requirement to a set of concrete alternative design mechanisms using a chain of rules with the category of “NF–REF”, “NF–OP”, “F–REF”, and “F–OP”. The refinement procedure is composed of one or more refinement steps. The general form of a refinement step is the reverse of the implication rules developed in the previous steps:

$$Term_{\ n} \xleftarrow{Rule\ Type} (Term_{\ 1}, \dots, Term_{\ m})$$

Each refinement step has two parts: (1) identifying the term to be refined next, (2) finding the rules whose consequent match the identified term. The decision about which term should be refined next is up to the human user. Since there is at most one rule whose consequent matches a given term, at most one rule is found and retrieved. The pseudo code of refinement procedure is shown in Figure 14.

```
Procedure: Refine Term
Input: Rule Set RS, A given term $Term_i$, A Working Rule Set WS
Output: Updated Working Rule Set WS

  for Rule r in RS
      if r's consequent is equal to $Term_i$ and r.category is not equal to "COR" and r is not in WS
          add r to WS
```

Figure 14. The Pseudo Code of Refinement Step

Input	**Output**
1- Security [API]	2- Security [API] $\xleftarrow{and}$ (Confidentiality [API], Privacy [API], Operational Security [API], Reliability [API])
3- Confidentiality [API]	4- Confidentiality [API] $\xleftarrow{and}$ (Message Confidentiality [API], Access Confidentiality [API])
5- Access Confidentiality [API]	6- Access Confidentiality [API] $\xleftarrow{Help}$ (Access Control [API])
7- Access Control [API]	8- Access Control [API] $\xleftarrow{and}$ (Access Authorization [API], Key and Certificate Management [ ])
9 - Access Authorization [API]	10- Access Authorization [API] $\xleftarrow{xor}$[2] (API-Key [ ], Username and Password [ ], Mutual Certificate-Based Authentication X.509 [ ], Open Authorization Version 2.0 [ ], Open-ID Connect Version 1.0 [ ])

Figure 15. An Instance of Applying the Refinement Procedure to Refine a Requirement

---

[2] To simplify the implementation of inference procedure, we have replaced the original OR relationship between design alternatives for access authorization to API to XOR.

The output of each refinement step is a set of non-functional or functional requirements decomposing the input requirement into more fine-grained ones using an “NF–REF” or “F–REF” rule, or a set of design mechanisms addressing a non-functional requirement or design objective using an “NF– OP” or “F–OP” rule. Refinement steps continue until a rule of category “F-OP” is found (i.e., design alternatives addressing the input requirement are found). Figure 15 shows an instance of applying the refinement procedure. The input to the procedure is security of API identified as a term in the form of “Security [API]”. The final output is five design alternatives for authorizing access to API.

4.2 The Stepwise Analysis and Evaluation Procedure

The second component of inference is a stepwise analysis and evaluation procedure. This procedure allows to analyze and evaluate the design alternatives identified at the end of refinement step against all the requirements identified in the requirements specification. The analysis and evaluation procedure has three steps: First, the rules with the rule category of “*COR*” are found and used to identify the trade-offs of all the design techniques. Second, the identified requirements are used to prune the retrieved rule set. Finally, the NFR forward evaluation procedure (Appendix 2) is used to evaluate the satisfaction of the identified constraints by selecting each of the design techniques.

4.2.1 The Analysis Procedure

The analysis procedure identifies the trade-offs of each design alternative identified at the end of refinement step. For this purpose, rules with the category of “COR” whose precedents are the identified design alternatives are retrieved and added to the chain of rules formed in the refinement procedure. Unlike the refinement procedure in which at most one rule matches a given term, there can be several “COR” rules that match a term representing a design technique.

The analysis procedure has two steps: (a) adding the correlation rules related to the terms representing the design techniques; (b) adding the terms reachable from the consequents of the added correlation rules. The reason for performing the second step is that some new terms (i.e., non-functional requirements) may be present in the consequents of the added “COR” rules that are related to the constraints identified in the requirements specification and need to be considered in the evaluation of the design techniques. The pseudo code of adding correlation rules is shown in Figure 16 and the pseudo code of adding reachable rules from correlation rules is shown in Figure 17.

```
Procedure: Add Correlation Rules
Input: Rule Set RS, A given term $Term_i$, A Working Rule Set WS
Output: Updated Working Rule Set WS
  for Rule r in RS
      if r's precedent is equal to $Term_i$ and r's category is equal to "COR"
          add r to WS
```

Figure 16 The Pseudo Code of an Analysis Step – Adding Correlation Rules Related to a Term reffering to a Design Technique

```
Procedure: Find Reachable Rules
Input: Rule Set RS, A Working Rule Set WS, A given term $Term_i$
Output: Updated Working Rule Set WS
  matchedRules = {}
  for Rule r in RS
      if (r's precedent is equal to $Term_i$ and r is not in WS)
          add r to matchedRules
  add matchedRules to WS
  for Rule r in matchedRules
      find Reachable Rules (RS, WS, r's consequent)
```

Figure 17. The Pseudo Code of Analysis Step – Adding Reachable Rules from a Given Term

An instance of the analysis procedure is illustrated in Figure 18. The input to the procedure is the term “API-Key []”, one design technique for authorizing access to an API identified at the end of the refinement step. The first output of the analysis step is six correlation rules identifying the impact of “API-Key” on some non-functional requirements. The second output is the rules reachable from the consequent of the added correlation rules.

Input	**Output**
API-Key [ ]	**Step1: Adding correlation rules related to API-Key** 1- (API-Key [ ]) $\xrightarrow{Make}$ Access Simplicity [API] 2- (API-Key [ ]) $\xrightarrow{Make}$ Usage Simplicity [API] 3- (API-Key [ ]) $\xrightarrow{Make}$ Latency [API] 4- (API-Key [ ]) $\xrightarrow{Help}$ Access Confidentiality [API] 5- (API-Key [ ]) $\xrightarrow{Break}$ Message Confidentiality [API] 6- (API-Key [ ]) $\xrightarrow{Break}$ Privacy [API]
	Step 2: Adding new rules whose precedents appear in the consequent of the added correlation rules 7- (Response Time [API], Latency [API], Throughput [API], Availability [API]) $\xrightarrow{and}$ Performance [API] : NF-REF 8- (Access Simplicity [API], Access Duration [API], Access Rate [API]) $\xrightarrow{and}$ Accessibility [API] : NF-REF 9- (Understandability [API], Efficiency [API], Usage simplicity [API], Consistency [API]) $\xrightarrow{and}$ Usability [API]: NF-REF

Figure 18. An Instance of Applying Analysis Procedure to Add Correlation Rules and Rules Related to Correlation Rules

4.2.2 The Pruning Procedure

After the step-wise refinement and analysis, a set of rules are retrieved for a given input term. The rules identify the design techniques that address the input requirement and their trade-offs. However, it is possible that some of the retrieved rules or some of the terms in the retrieved rules are irrelevant to a given design case. Thus, the set of retrieved rules need to be pruned to eliminate unrelated rules and terms. An instance of unrelated rules and terms in the retrieved rule set for the “Account API” is identified in Figure 19. The relevance of the retrieved rules and terms is decided based on the “Account API” requirements specification shown in Figure 1.

Initial Input	**Retrieved and Added Rules**
Security [API]	1- (Confidentiality [API], Privacy [API], ~~**Operational Security [API]**~~, ~~**Reliability [API]**~~) $\xrightarrow{and}$ Security [API]: NF-REF 2- (Message Confidentiality [API], Access Confidentiality [API]) $\xrightarrow{and}$ Confidentiality [API]: NF-REF 3- (Access Control [API]) $\xrightarrow{Help}$Access Confidentiality [API]: NF-OP 4- (Access Authorization [API], ~~**Key and Certificate Management [ ]**~~) $\xrightarrow{and}$ Access Control [API]: F-REF 5- (API-Key [ ], Username and Password [ ], Mutual Certificate-Based Authentication X.509 [ ], Open Authorization Version 2.0 [ ], Open-ID Connect Version 1.0 [ ]) $\xrightarrow{xor}$ Access Authorization [API]: F-OP 6- ~~**(API-Key [ ]) $\xrightarrow{Make}$ Access Simplicity [API]: COR**~~ 7- ~~**(API-Key [ ]) $\xrightarrow{Make}$ Usage Simplicity [API]: COR**~~ 8- (API-Key [ ]) $\xrightarrow{Make}$ Latency [API]: COR 9- (API-Key [ ]) $\xrightarrow{Help}$ Access Confidentiality [API]: COR 10 -(API-Key [ ]) $\xrightarrow{Break}$ Message Confidentiality [API]: COR 11- (API-Key [ ]) $\xrightarrow{Break}$ Privacy [API]:COR 12- (~~**Response Time [API],**~~ Latency [API], ~~**Throughput [API]**~~, ~~**Availability [API]**~~) $\xrightarrow{and}$ Performance [API]: NF-REF 13- ~~**(Access Simplicity [API], Access Duration [API], Access Rate [API]) $\xrightarrow{and}$ Accessibility [API] : NF-REF**~~ 14-~~**(Understandability [API], Efficiency [API], Usage simplicity [API], Consistency [API]) $\xrightarrow{and}$ Usability [API]: NF-REF**~~

Figure 19. An Instance of Pruning the Rule Set after the Analysis Step – Unrelated Terms and Rules are Crossed Out.

Unrelated rules can remain in the retrieved rules set and be evaluated. However, their evaluation does not impact the selection procedure since the selection procedure only considers the constraints identified in the input requirement specification. However, unrelated terms need to be pruned. Unrelated terms are the terms that are not reachable from the identified design techniques (i.e., the rule(s) with the category of "F–OP"). To prune the unrelated terms, their existence should be neutralized in the evaluation of the rules. For this purpose, the rule type of the rules in which the unrelated terms exist is checked. If the rule is of type "AND", and the unrelated term is in the precedent of the rule, the satisfaction value of the term is set to "Satisficed" to neutralize the impact of the term on the evaluating the rule. If the rule is of type "OR" and the term appears in the precedent of the rule its satisfaction value is set to "Denied". If the term is unrelated and it does not appear in the precedent of any rule, it can be evaluated. However, it does not impact the selection procedure.

(1) API-Key [ ] := Sat
(API-Key [ ], Username and Password [ ], Mutual Certificate-Based Authentication X.509 [ ], Open Authorization Version 2.0 [ ], Open-ID Connect Version 1.0 [ ]) $\xrightarrow{xor}$ Access Authorization [API]
(API-Key [ ]) $\xrightarrow{Make}$ Latency [API]
(API-Key [ ]) $\xrightarrow{Help}$ Access Confidentiality [API]
(API-Key [ ]) $\xrightarrow{Break}$ Message Confidentiality [API]
(API-Key [ ]) $\xrightarrow{Break}$ Privacy [API]

(2) Access Authorization [API] := Sat
Latency [API] := Sat
Access Confidentiality [API]:= PSat
Message Confidentiality [API] := Den
Privacy [API] :=Den
(Message Confidentiality [API], Access Confidentiality [API]) $\xrightarrow{and}$ Confidentiality [API]
Latency [API] $\xrightarrow{Make}$ Performance [API]

(3) Confidentiality [API]:= Den
Performance [API]:= Sat
(Confidentiality [API], Privacy [AP]) $\xrightarrow{and}$ Security [API]

(4) Security [API]:= Den

∴ Confidentiality [API] := Den, Privacy[API] :=Den, Latency [API]:= Sat

Figure 20. An instance of the step-wise forward evaluation procedure

4.2.3 The Step-Wise Evaluation Procedure

To evaluate the impact of each identified design alternative against the non-functional requirements, the pruned rule set is evaluated using the NFR forward evaluation procedure (Appendix 2). The evaluation results provide a comparison basis to select the most appropriate design alternative from among the introduced designs. For this purpose, the terms in the pruned rule set are evaluated starting from the fact that the term related to each design technique (i.e., the terms appearing in the precedent of an "F–OP" rule) is selected (i.e., the term is fully satisficed.). The pseudo code of the evaluation step is shown in Figure 21. An instance of the step-wise evaluation procedure is shown in Figure 20, evaluating the pruned rule set retrieved for the term "Security [API]", starting from the term "API-Key []".

Procedure: Evaluate the Working Rule Set Starting from a Given Term
Input: Working Rule Set *WS*, A given $Term_i$
Output: Evaluated Working Rule Set

```
currentTerm = $Term_i$
currentTerm's SatisfactionValue = Sat
Queue NextRules = {}
do
    for Rule r in WS
        if r's precedent is equal to currentTerm
            NFR-Evaluate (r) [3]
            enqueue r in NextRules
    if (NextRules is not empty)
        nextRule = dequeue NextRules
        currentTerm = nextRule's consequent
    else currentTerm = null
while (currentTerm is not null)
```

Figure 21. The Pseudo Code of the Evaluation Step

4.3 The Selection Procedure

The third component of inference is a selection procedure to choose the most appropriate design alternatives for a given requirements specification. The most appropriate design alternatives are the alternatives that fully or closely meet all the constraints (i.e., requirements) identified in the specification. To identify the best designs, alternatives are scored against the requirements specification, and then the ones with the highest score are selected.

To score the design alternatives, each requirement in the specification is represented as a triple with the general form of $\langle Contraint, Priroity, Satisfaction\ Value\rangle$. The satisfaction values of the constraints are calculated in the step-wise analysis and evaluation procedure. To calculate a numerical score, we map the satisfaction values to integer values using the following mapping: {*Den* → -2, *PDen* → -1, Conflict → 0, Unknown → 0, *PSat* → +1, *Sat* → +2}. The conflict and unknown values can also be replaced with other satisfaction values prior to calculating the score. We also assume that the requirements are associated with priorities belonging to the set of {*Low*, *Normal*, *High*}, with the ordering of $Low < Normal < High$. We map these priorities to integer weights as follows: {Low → 1, *Normal* → 2, High → 4}. To calculate the score of each alternative, the numbers related to the satisfaction of the constraints in the alternative are multiplied by the weight of the constraints and summed up using the following formula:

$$Score\ (design\ alternative\ i) = \sum_{j=1}^{n} weight\ (constraint\ j) * sat\ (constraint\ j, design\ alternative\ i)$$

, n is the number of constraints identified in the requirement specification.

To select the best designs, the score of design alternatives are compared against the score of an ideal design that fully meets specified requirements, and the designs with the closest score are selected. To calculate the score of an ideal design, the priority of the constraints in the requirement specification are multiplied by +2 (denoting that the constraints are expected to be fully satisficed), and then summed up as follows:

$$Score\ (ideal\ design) = 2 * \sum_{j=1}^{n} weight\ (constraint\ j)$$

The most appropriate design alternatives are the alternatives with the minimum difference from the above score:

$$\text{selected design alternatives} = \underset{1 \le i \le m}{\arg\min}(Score\ (ideal\ design) - Score\ (design\ alternative\ i))$$

, m is the number of design alternatives identified at the end of step-wise refinement procedure.

---

[3] Appendix 2 explains how to evaluate an implication rule in the NFR multi-valued logic.

Table 9 shows an instance of applying the selection procedure to select from among the five design alternatives identified at the end of the refinement procedure to address security of API.

Table 9. An Instance of the Selection procedure – Selected Designs are Underlined and Bolded.

Requirements Specification	**Constraint**	**Confidentiality [API]**	**Privacy [API]**	**Latency [API]**	**Score**
	Priority	**High**	**High**	**Medium**	
	Expected Satisfaction Value	**Sat**	**Sat**	**Sat**	**20**
Available Design Techniques	API-Key	Den	Den	Sat	-12
	Username and Password	**PSat**	**PSat**	**PDen**	**6**
	Mutual Authentication	Sat	Den	PDen	-2
	OAuth 2.0	**PSat**	**PSat**	**PDen**	**6**
	OpenID Connect	PSat	PSat	Den	4

5 Implementation

We have developed Rational API Designer (RAPID), the working prototype of a tool that uses the developed rules and follows the step-wise inference procedure to assist its users with designing APIs. The source code of RAPID is available on GitHub[4] under Creative Commons license.

RAPID is designed and implemented as a rule-based knowledge-based system in Java and has four components: (a) a rule base, (b) a rule explorer (c) a user interface; and (d) a reasoner, as shown in Figure 22. The "Rule Base" stores and manages the rules encoding design knowledge, and has two components: (a) A "Rule Base" file, and (b) a "Rule Base Manager". "Rule Base" file stores the rules. Currently, there are 156 design rules in RAPID as listed in Appendix 3. The responsibilities of the "Rule Base Manager" include: (a) loading the rules from the file into memory, (b) building an in-memory version of the rule base, and (c) finding and retrieving rules related to a user's query. The rule base manager collaborates with the "Rule Explorer" component.

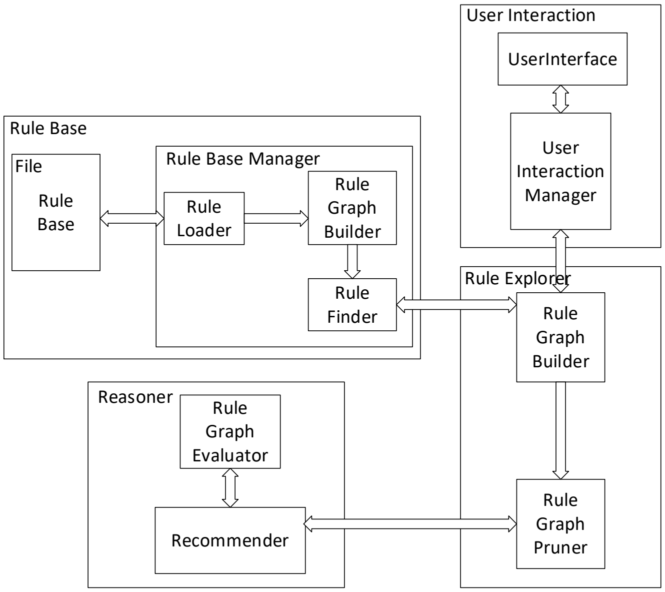


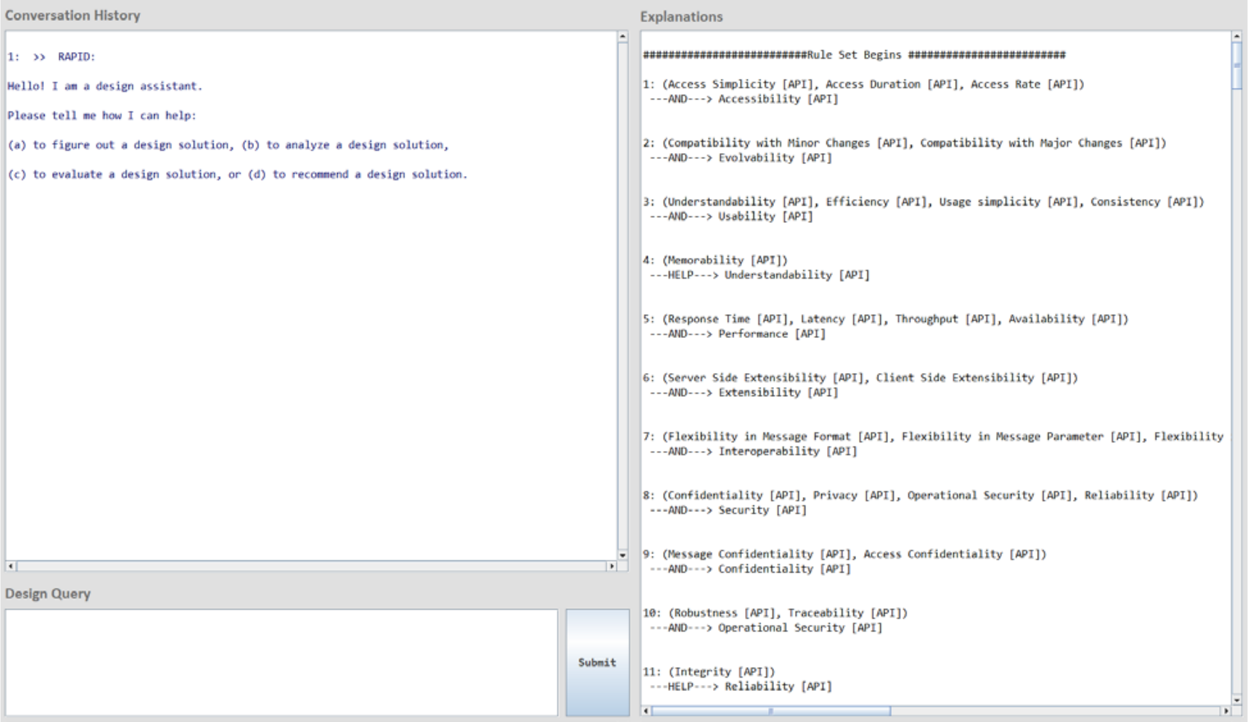
Figure 22. The High-Level Architecture of RAPID

The "Rule Explorer" retrieves the rules relevant to a session of interactions with a user from the rule base and manages the retrieved rules until the interaction session ends. For this purpose, the explorer receives user's query from "User

---

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

Interaction Manager”, and interacts with the “Rule base Manager” to respond to the user’s query. “Rule Explorer” also builds and prunes the “Working Rule Set” (i.e., the retrieved rules related to a session of interactions with the user) and sends the “Working Rule Set” to the “Reasoner” component for evaluation and recommendation.



```
1:  >>  RAPID:

Hello! I am a design assistant.

Please tell me how I can help:

(a) to figure out a design solution, (b) to analyze a design solution,

(c) to evaluate a design solution, or (d) to recommend a design solution.
```

```
##########################Rule Set Begins #########################

1: (Access Simplicity [API], Access Duration [API], Access Rate [API])
 ---AND---> Accessibility [API]


2: (Compatibility with Minor Changes [API], Compatibility with Major Changes [API])
 ---AND---> Evolvability [API]


3: (Understandability [API], Efficiency [API], Usage simplicity [API], Consistency [API])
 ---AND---> Usability [API]


4: (Memorability [API])
 ---HELP---> Understandability [API]


5: (Response Time [API], Latency [API], Throughput [API], Availability [API])
 ---AND---> Performance [API]


6: (Server Side Extensibility [API], Client Side Extensibility [API])
 ---AND---> Extensibility [API]


7: (Flexibility in Message Format [API], Flexibility in Message Parameter [API], Flexibility
 ---AND---> Interoperability [API]


8: (Confidentiality [API], Privacy [API], Operational Security [API], Reliability [API])
 ---AND---> Security [API]


9: (Message Confidentiality [API], Access Confidentiality [API])
 ---AND---> Confidentiality [API]


10: (Robustness [API], Traceability [API])
 ---AND---> Operational Security [API]


11: (Integrity [API])
 ---HELP---> Reliability [API]
```

Figure 23. The User Interface of RAPID

The “User Interaction” component interacts with the user in natural language, and allows the user to pose design queries over the rule base and to receive a set of responses. This component includes a “Graphical User Interface” (GUI) and a “User Interaction Manager”. The GUI is responsible for receiving the input of the user, sending it to “User Interaction Manager” for further processing, and returning the responses of the system to the user. RAPID’s GUI has been designed to be interactive (similar to the interface of a chat bot). The consultation interface has three components: (a) a “Conversation History”, (b) a “Design Query”, and (c) an “Explanation”. “Conversation History” keeps the history of interactions between the system and the user. “Design Query” is where the user inputs her questions and receives responses. RAPID knows five user commands of “figure out”, “analyze”, “evaluate”, “recommend”, and “exit”. The topic of the design queries should be entered in the general form of “Type [Topic]” in the design query section. The input keywords are matched to the precedent and consequents of the rules available in the rule base. “Explanations” section is where the design assistant (RAPID) provides further explanations about its line of reasoning. When RAPID program starts to run, the “Explanations” component shows all the rules available in the rule base. A snapshot of the user interface when the tool starts running is shown in Figure 23. The explanation section also shows the steps of reasoning and selection procedure.

The “User Interaction Manager” is responsible for receiving the input query from the user interface, processing the query, and passing the input to the “Rule Explorer”, where the rule base is explored. The user interaction manager is also responsible for receiving the response of the “Rule Explorer” and returning it to the user via the user interface.

The “Reasoner” component is responsible for evaluating the rules related to a session of interactions with a user (i.e., the “Working Rule Set”) and selecting and recommending a set of design alternatives to her. “Reasoner” has two components: (a) An evaluator which is responsible for evaluating the working rule set and returning the evaluation results. Currently, the evaluator implements a subset of the NFR forward-chaining procedure for the rule sets which do not require conflict resolution. (b) A recommender which is responsible for receiving a set of requirements from the user and returning design the alternatives that best match the input requirements. For the sake of simplicity, the recommender component currently supports “XOR” relationships between design alternatives. Thus, we have altered “OR” rules encoding the knowledge related to design alternatives to “XOR” when inserting the rules in RAPID’s rule base. The altered rules types are added in parenthesis in the rules listed in Appendix 3.

6 Evaluation

To evaluate the validity of the design recommendations provided by RAPID, we have measured the accuracy of recommendations; i.e., the ratio of acceptable recommendations to the total number of recommendations:

$$Accuracy = \frac{\#\ Acceptable\ Answers}{\#\ Answers}$$

To identify the acceptability of an answer, we have used a consensus building and voting procedure [34], an approach commonly used in decision making procedures. We have considered a recommendation as acceptable for a given design case if it is considered acceptable by the majority of a group of domain experts.

To assess the accuracy of the design recommendations as defined above, we have performed three main steps: (1) We have prepared a design exam containing seven API design specifications for various software platforms and thirty questions asking to design a specific non-functional or related functional requirement for the specified APIs. We have consulted RAPID to arrive at an answer for each design question. We have prepared an API design exam notebook containing the API design specifications, the design questions, the answers generated using RAPID, and a scheme for evaluating the answers. (2) We have asked seven well-experienced software architects and developers to blindly evaluate the answers of RAPID in the exam. (3) We have analyzed the results of the evaluation both quantitatively and qualitatively. In the following, we describe the details of each evaluation step.

6.1 The API Design Exam Notebook

In the first step of evaluation, we prepared an API design exam notebook. The design exam notebook contained seven API design specifications for various software platforms, and thirty questions asking about designing a specific functional or non-functional requirement for the specified APIs. To prepare the API design exam notebook, seven steps were performed: (1) collecting some API design cases, (2) selecting from among the selected cases, (3) writing API design specifications, (4) assigning some design questions to each API specification, (5) consulting RAPID to arrive an answer for each question, (6) providing a scheme for evaluating the answers, (7) preparing an exam notebook to communicate it with the domain experts.

6.1.1 Collecting API Design Cases

The API design specifications were collected and compiled from real-world API specifications and documentation of various software platforms available on the Web. For this purpose, we searched the Web for the web-sites of some software platforms which had exposed a set of public or protected APIs towards public developers or partners. During the search, we could find numerous software platforms exposing open and protected APIs towards developers. There were also well-documented web pages about the specification of the exposed APIs. In most of the cases, there were more than one platform offering the same type of services and APIs. In such cases, we studied the web-site of two or three platforms to gather, compile, and infer the required information for preparing a final design specification.

6.1.2 Selecting API Design Cases

To select from among the available API specifications available on the web, we considered the following criteria:

- *Covering various kinds of software systems and services for which an API is to be developed:* To gather a representative sample of the design cases that could be consulted with RAPID, we selected platforms and APIs such that they cover a wide-variety of software systems [35], including e-commerce platforms, cloud platforms, safetycritical and embedded platforms, enterprise platforms, and big data platforms. This variety helps show the applicability of the developed framework in various application domains and test the validity of recommendations for diverse design cases.

- *Covering various categories of non-functional requirements with different priorities:* In each type of software system, some categories of non-functional requirements have higher priority than others. For example, in a banking platform, security requirements such as confidentiality of customers' data are very critical, whereas in a cloud platform, performance requirements, such as real-time response time and low latency are very critical. To cover all the distinct cases of the input domain, we selected the platforms and the APIs such that they cover a wide range of non-functional requirements with different priorities.
- *Covering various types of APIs*: We selected examples to cover all the distinct cases of the input domain, including (a) open APIs (public APIs), (b) private APIs, and (c) protected or partner APIs.

Considering the above criteria, seven API design cases were selected from six different software platforms: (1) "a Banking Platform – The Payment API", (2) "a Weather Map Platform – The Current Weather API", (3) "a Cloud Storage Platform – The Write Bucket / Object API", (4) "a Vehicle Control Platform – The Lock Status API", (5) "a Flight Data Platform – The Flight Time Table API", (6) "a Social Network Platform – The Friends API", and (7) "a Social Network platform – The Filter and Track Posts API".

6.1.3 Writing API Design Specifications

For each selected API, a design specification containing the following information was prepared: (a) the specification of the API and some of its use cases, (b) The type of the API (i.e., public or open, protected, or private API), (c) the functionality of the API and the API requests and responses, and, (d) The non-functional requirements and their priority in the API design case. The specifications were simplified to provide an overall sense of the requirements that should be considered in developing the selected APIs.

6.1.4 Assigning Design Questions to Each API Specification

We developed a question bank covering all the topics (i.e., non-functional requirements and related design objectives) for which RAPID can provide design guidelines (see Appendix 4). The question bank contains example design questions for each topic. We then distributed the design questions over the seven selected API design specifications. To distribute the questions over the design cases, we considered the following criteria:

- *Covering most of the topics for which RAPID can provide design assistance at least once*. This strategy helps examine the breadth of knowledge, and check applicability and relevance of the suggestions for different cases.
- *For the topics that RAPID can provide a final recommendation (from among several design alternatives), we asked them at least twice for two different cases*. This strategy helps examine the depth of design knowledge, to check the accuracy of the recommendations for the same questions over different cases, and to show that RAPID does not provide the same answer for different cases.
- *For the topics that RAPID can provide some design guidelines but does not have enough knowledge to recommend a final design, we ask them at least once*. For some functional and non-functional requirements, RAPID only provides some alternative designs (without recommending one) or some design objectives without concrete alternatives. We selected some questions to cover these topics at least once.

We prioritized the design cases based on their non-functional requirements and then assigned each question to one or two design cases for which the asked non-functional requirement or design objective had the highest priority and relevance. Overall, thirty design questions were distributed among the seven selected API requirements specifications.

6.1.5 Consulting RAPID to Receive an Answer

We consulted each of the thirty design questions with RAPID to arrive at some final recommendations or guidelines. For this purpose, we provided the topic of each design question as input to RAPID in the form of "*Type [Topic]*". We interacted with RAPID up to the point that a set of design guidelines or alternatives were recommended. We recorded the responses and recommendations of RAPID to each design question.

6.1.6 Scheme for Evaluating the Design Recommendations

To design the evaluation scheme, we considered two points: (a) There is often more than one suitable design for a given requirements specification and a given question. (b) Different domain experts often arrive at different designs for the same design question. Thus, we designed six ratings for evaluating the answers:

a) "*Acceptable Identical*": The answer is the same as what the evaluator would recommend for the design case.

b) "*Acceptable Alternative*": The answer is different from the recommendation of the evaluator but is an acceptable alternative for the evaluator.

c) "*Partly Unacceptable*": The answer is partly invalid or incomplete.

d) "*Completely Unacceptable*": The answer is completely invalid or irrelevant to the case.

e) "*Others*": There is not sufficient information about the design case to decide, and

f) "*Others*": The evaluator does not know the answer to the question or does know about the provided answers.

In the evaluation instructions, we asked the domain experts to follow three steps: (1) to study each API design specification and the related design questions carefully, (2) to think of their own answers to the design questions, and (3) to evaluate the validity of the provided answers. We also asked the evaluators to identify the invalid parts of an answer if their evaluation of an answer was "*unacceptable – partly incomplete or irrelevant.*", and to briefly write down the answers that they would suggest for the question, in the comment section provided at the end of each answer.

6.1.7 Preparing the Exam Notebook

We prepared a design exam notebook, containing the API design specifications, the design questions, and the answers that were provided using RAPID (see Appendix 5). The answers were typed in the form of short answers such that the domain experts would no recognize that they are provided by a tool. To avoid ambiguity and misunderstanding, we added a brief description to some answers explaining what some answer means. The exam notebook also explained briefly the objective of the study without disclosing how the answers were provided, contained a set of instructions for the evaluators, the estimated time of the study, and a form containing some multiple-choice question for collecting the background and experience of the evaluators in software development and in API development in particular.

6.2 Evaluating the Design Recommendations

To evaluate the answers provided by RAPID, we recruited a group of seven domain experts. Five of the recruited domain experts were software engineers, architects, or researchers working in two companies of Autodesk and Google. Two of the evaluators were researchers and practitioners affiliated with the University of Toronto. From the seven recruited evaluators, six of them had more than ten years of professional experience in software development. Moreover, six of the evaluators had more than three years of dedicated experience in web, cloud, or micro-service development and had participated in more than five related projects. Four of the evaluators had also some years of related research experience in addition to their practical experience. Table 10 shows the anonymized profile of the evaluators.

We distributed the exam notebook among the domain experts via e-mail, asking them to return their evaluation results within two weeks after the receipt of the notebook. To avoid evaluation bias, we created a blind evaluation environment, and we did not disclose that the answers were provided by a software system until the end of the study.

Table 10. The Profile of the Domain Experts

Evaluator	*Evaluator 1*	*Evaluator 2*	*Evaluator 3*	*Evaluator 4*	*Evaluator 5*	*Evaluator 6*	*Evaluator 7*
Profession	Researcher	Practitioner	Practitioner and Researcher	Practitioner	Practitioner	Practitioner and Researcher	Practitioner and Researcher
Software and System Development Experience	more than 10 years	more than 10 years	more than 10 years	more than 10 years	more than 5 years	more than 10 years	more than 10 years
Web, Cloud, Micro-Service Experience	more than 3 years	more than 3 years	more than 3 years	more than 3 years	more than 3 years	less than 3 years	more than 3 years
Number of Involved Web, Cloud, or Micro-Service Projects	more than 5 projects	more than 5 projects	more than 5 projects	more than 5 projects	more than 5 projects	less than 5 projects	more than 5 projects
Received Formal Education on Web, Cloud, or Micro-Service	No	Yes	No	Yes	Yes	Yes	Yes
Research on Web, Cloud, Micro-Service	more than 3 years	No	less than 3 years	No	more than 3 years	No	less than 3 years
Self-Assessed Confidence and Knowledge in API, Web-Service, Cloud, or Micro-Service Development	Experienced	Experienced	Experienced	Expert	Experienced	Familiar	Expert

6.3 Analyzing the Evaluation Results

The evaluation results were in two forms: (1) the ratings given to the answers, (2) the comments provided by the evaluators supporting the given ratings. The ratings of the answers are provided in Appendix 5. The comments of the evaluators are available in [36].

6.3.1 Analyzing the Ratings of the Answers

In total, 210 ratings were given by the seven evaluators to the 30 answers (See Appendix 5). From these 210 ratings, 85 were "*Acceptable – Identical*", 47 were "*Acceptable – Alternative*", 46 were "*Partly Unacceptable*", 15 were "*Completely Unacceptable*", 7 were "*Unable to decide based on the specification*", and 10 were "*Do not know the answer*". The distribution of the ratings is shown in Figure 24.

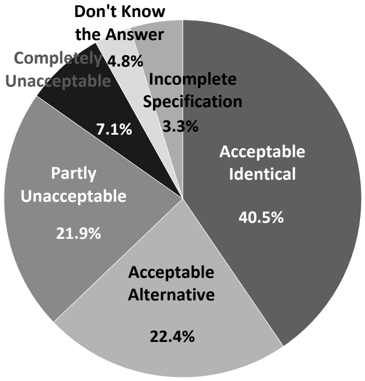


Figure 24. The Distribution of the Ratings that RAPID's Answers Received in the Evaluation (N=210)

To measure the accuracy of the provided answers, we considered a recommendation as acceptable if the majority of the experts had given an "*Acceptable – Identical*" or "*Acceptable – Alternative*" rating to the answer. In our study, majority means at least four out of the seven evaluators. From among the thirty provided answers, twenty-two of them were considered as acceptable by the majority of evaluators while eight of the answers were not (see Table 11).

Table 11. Obtained Accuracy in the Evaluation Study

Number of Acceptable Answers	Total Number of Answers	Percentage of Acceptable Ratings from the Majority of Evaluators (N=7)
22	30	73.3%
An answer is considered as acceptable if it had received either an "*Acceptable – Identical*" rating or "*Acceptable - Alternative*" rating from at least four out the seven evaluators.		

Out of the eight answers not receiving a majority acceptance rating, only three of them were rejected by the majority of the evaluators (i.e., receiving a "*partly-unacceptable*" or "*completely-unacceptable*" rating from at least four of the evaluators). Four of the answers received an equal number of acceptable and unacceptable ratings, and the reason for not receiving a majority acceptance was that there was at least one evaluator who did not know the answer or could not decide about the validity of the answer based on the specification. One of the answers had received more acceptable ratings than unacceptable ones, but the number of acceptable ratings didn't meet the majority criteria since there were three evaluators who did not know the answer or could not decide about the validity of the answer based on the specification. The overall distribution of the ratings given to the eight unaccepted answers is shown in Figure 25.

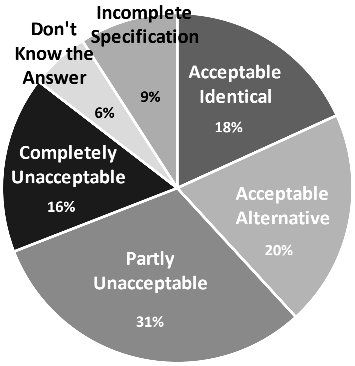


Figure 25. The Overall Distribution of the Ratings for the Eight Unaccepted Answers (N=56)

6.3.2 Analyzing the Comments Supporting the Given Ratings

Six out of the seven evaluators had supported their evaluation of each answer with some comments especially if their rating was not "*acceptable-identical*". The only evaluator who had not provided any comments had considered 29 out of 30 answers as "*acceptable-identical*" or "*acceptable-alternative*". Overall, out of the 210 ratings given by all the seven evaluators, 135 of them were supported by some comments. These comments are available in [36]. The comments were of three categories: (a) the problems with the provided answers, (b) the suggested answers of the evaluators for the design questions, and (c) both the problems and the suggested answers of the evaluators. We used the comments of evaluators for three purposes: (1) case-by-case analysis of the eight answers that did not receive a majority acceptable rating. (2) understanding why some answers had been considered as unacceptable by some of the evaluators; and (3) understanding the overall differences between the guidelines of RAPID and the recommendation of the evaluators in case an "*acceptable-alternative*" rating was given to an answer.

6.3.2.1 Case by Case Analysis of the Evaluators' Comments about the Eight Unaccepted Answers

We analyzed all the comments supporting the ratings given to the eight answers that did not receive a majority acceptance vote case by case. Our research question while analyzing these answers was as follows:

- *RQ 1. "What is the problem with the unaccepted answer and what is the appropriate answer?"*

The reasons of the domain experts for not accepting these eight answers, or the suggested answers to the related questions varied and there was no agreement on the appropriate answers. Thus, we could not reach to a case-by-case conclusion about the problems with these answers. However, two reasons were common in the comments:

1. *The design guidelines were necessary but incomplete*: This problem was in part due to the incompleteness of the design knowledge encoded in RAPID. The incompleteness related to the lack of design techniques to refine a design objective and lack of relationships between some design techniques and design objectives that were encoded in RAPID. Moreover, to answer some design questions, case-specific details were required unrelated to the design of the web API, and the question topic, but required to answer the complete question designed to ask about a topic.
2. *The answers, the design questions, and sometimes the relevance of an answer to the question topic were misunderstood by the evaluators*: This was in part due to the design of the exam notebook and the ambiguity of some design questions, and in part due to the short answers that RAPID provides without explaining the answer.

6.3.2.2 Analyzing Evaluators’ Comments Supporting the Unacceptable Ratings

Out of the 210 ratings given by the seven evaluators, 61 of them were either “*unacceptable - partly incomplete or irrelevant*” or “*unacceptable - completely unacceptable or irrelevant*”. From these 61 ratings, 60 of them were supported by some comments. We analyzed these comments to identify the overall reasons for the unacceptability of the answers for some evaluators. Our research question while analyzing these comments was as follows:

- *RQ 2. “What is the overall reason for considering the answer as unacceptable?*”

To answer the above question, we performed open coding [37, 38, 39]. We coded and categorized each comment with one or two labels identifying the problem with the answer, and one or two reasons for rejecting the answer. We assigned the labels and reasons based on explicit or implicit clues in the comments of the evaluators. Explicit clues were obtained based on exact parts of the comments explicitly referring to a reason for the rejection. Implicit clues were inferred by analyzing the relationship between the comment, the answer, and the topic of a question. The coded comments are available in [36]. Two instances of the evaluators’ comments, and the assigned labels and reasons for rejection are shown in Table 12.

Table 12. Two Instances of Evaluators’ Comments Supporting Unacceptable Ratings and the Assigned Codes

API Design Specification: The Payment API
Question Number: 1
Question Topic: Access Authorization [API]
Answer: “Username Password XOR Open Authorization Version 2.0 XOR OpenID Connect Version 1.0”

Evaluator’s Comment	Assigned Code
Rating: “The mentioned design(s) are partly invalid or incomplete.” **Comment**: “Username and password **are the base information to authenticate (not limited)**. OpenID is a protocol for authentication and a component of OAuth 2.0 communication protocol for authorization. So not a "OR" but a "AND".”	**Label 1**: The evaluator is explicitly stating that the answer is incomplete. **Label 2**: The evaluator is stating that “AND” should be used instead of “OR”. However, the statement is not valid. Open ID is built on top of OAuth and these two can be used separately for different purposes. It seems that the evaluator has misunderstood the answer. **Category - Reason for not accepting the answer 1**: “One part of the answer is valid but incomplete.” **Category - Reason for not accepting the answer 2**: “One part of the answer is misunderstood.”
Rating: “The mentioned design(s) are partly invalid or incomplete.” **Comment**: “Check - These APIs are **too sensitive to be defined with such spartan requirements / designs**. Though the mentioned technologies are useful towards this end, **it is not enough to state that these will be useful for this implementation, where security and availability are so critical**.”	**Label**: The evaluator is stating that the suggested answers do not meet some requirements of the case (i.e., security and availability) and are not appropriate or ideal for the specific case. **Category - Reason for not accepting the answer**: “The answer is inappropriate for the case.”

After coding the comments, five common reasons emerged explaining the unacceptability of the answers:

1. "*Answer is valid but incomplete.*": the provided guideline or some parts of it needs further elaborations and addition of further details; or more design mechanisms are required to address the question.
2. "*Answer is inappropriate.*": the provided design guideline or some parts of it is inappropriate for the given case; or the provided design guideline is not an ideal solution for the given case and better solutions exist; or the suggested design guideline does not meet some requirements of the case.
3. "*Design question has been misunderstood.*": the question is misunderstood or differently understood by the evaluator; or the evaluator has some objections to the design question.
4. "*Answer has been misunderstood.*": the answer is misunderstood or differently understood by the evaluator.
5. "*The logical relationship between the answers is inappropriate.*": the logical relationship between the two or more answers for a design question is inappropriate where more than one design guideline were provided as an answer.

Out of the 68 identified reasons associated to 60 unacceptable ratings, 23 were the "*Answer is valid but incomplete.*". 15 were "*Answer is inappropriate.*". 9 were "*Question has been misunderstood*.". 17 were "*Answer is misunderstood.*". 4 were "*The logical relationship between the answers is inappropriate.*" The distribution of the reasons behind the rejection of the evaluators is summarized in Figure 26.

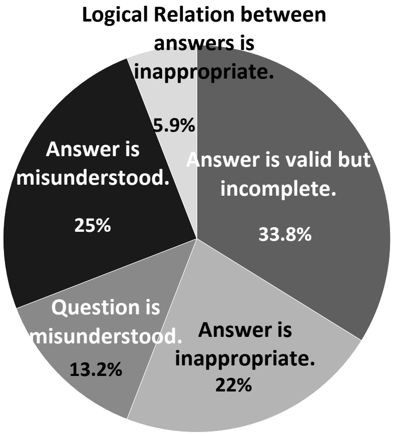


Figure 26. The Distribution of the Reasons for Considering Some Answers as Unacceptable (N=60)

Overall, out of the 60 comments supporting the unacceptable ratings, in 33.8 % of the cases, the reason for not accepting an answer was that the provided design guidelines were valid but incomplete. This incompleteness was partly due to the lack of enough API design knowledge available to RAPID, partly due to the lack of case-specific details in the answers which were not related to API design but related to the design of the back-end of the API, and partly due to lack of enough explanations and details about the answers. 22% of the answer seemed inappropriate or unideal to the evaluators. This inappropriateness was mainly because that an evaluator believed that the answer could not meet some requirements of the case. In 13.2% of the cases, the question was misunderstood and in 25% of the cases, the answer was misunderstood. These misunderstandings were in part due to the problems with the design of the exam notebook (the ambiguity of the designed questions), and in part due to the short answers that RAPID provides with no explanations about the definition of answers. Finally, 5.9% of the unacceptable ratings was because that the evaluator had considered the logical relationship between the answer should be different. Out of the four comments categorized as inappropriate logical relationships, one of them was related to altering "OR" relationship between design techniques to "XOR" in the rule base of RAPID, due to limitations in the implementation of the recommendation component.

The above insight suggests that statistically the inaccuracy of recommendations mainly rooted in three problems: (1) misunderstanding of some answers or design questions by the evaluators (38.2%), (2) valid but incomplete design guidelines (33.8%), and (3) inappropriate answers (22%).

6.3.2.3 Analyzing Evaluators' Comments Supporting "Acceptable-Alternative" Ratings

Out of the 210 ratings given by all the domain experts, 47 of the ratings were "*acceptable-alternative*". Out of these 47 ratings, 38 of them were supported by some comments. We studied these comments to gain insight into how the recommendations of the evaluators differ from the suggestion of RAPID. Our research question while studying the comments was as follows:

RQ 3. "*How do the recommendations of the evaluators differ from the design guidelines suggested by RAPID*?"

We coded the evaluators' comments supporting "*acceptable-alternative*" ratings to identify the overall difference between RAPID's and domain experts' recommendations. The coded comments are available in [36]. Two instances of the evaluators' comments and the assigned labels explaining the difference are shown in Table 13.

Table 13. Two Instances of Evaluators' Comments Supporting "Acceptable-Alternative" Ratings and the Assigned Codes

API Design Specification: The Payment API
Question Number: 1
Question Topic: Access Authorization [API]
Answer: "Username Password XOR Open Authorization Version 2.0 XOR OpenID Connect Version 1.0"

Evaluator's Comment	Assigned Code
Rating: "Acceptable – But I suggest a different design." **Comment:** "I'd suggest username password **with a required two-factor token."**	**Label: "**An Augmented Version of the Design Guideline."

API Design Specification: The Payment API
Question Number: 7
Question Topic: Failure Detection [API]
Answer: "Circuit Breaker XOR Response Time-Outs"

Evaluator's Comment	Assigned Code
Rating: Acceptable – But I suggest a different design. **Comment:** "I would incorporate *both* a and b..."	**Label: "**A modified Version of the Logical Relationship."

After coding the comments, six themes emerged explaining the difference between the recommendations of RAPID and the domain experts:

1. "*An augmented version of the design guideline*": The evaluators had elaborated and added further cases-specific or general details, or further mechanisms to RAPID's guidelines to completely answer the question.
2. "*A modified version of the design guideline – Logical combination*": The evaluators had modified the logical relationship between the design guidelines.
3. "*A pruned version of the design guideline"* The evaluators had eliminated one of the guidelines if more than one guideline was provided.
4. "*A different design guideline – Valid*" The evaluators had suggested a different set of guidelines as the answer.
5. "*A different design guideline – Misunderstanding the question*". The evaluators had suggested a different set of guidelines to answer the question due to a different understanding of the question, or due to having arguments about the design question, or due to answering a modified version of the question.
6. "*A different design guideline – Misunderstanding the answer*". The evaluators had suggested a different set of guidelines to answer the question due to a different understanding or incomplete understanding of the answer.

For the 38 "*acceptable-alternative*" ratings supported by some comments, 42 labels were identified explaining the difference between the recommendations. Some comments were associated with two labels. Out of these 42 identified recommendation differences, 14 were an augmented version of RAPID's guideline, 9 were a modified version of

RAPID's guideline (i.e., modifying the logical relationship), 2 were a pruned version of RAPID's guidelines (eliminating one design guideline) , 9 were suggesting valid and different guidelines from what RAPID had suggested, 3 were suggesting different guidelines due to a different understanding of the question or the API specification, and 5 were suggesting different guidelines due to a different understanding of RAPID's design guidelines. The distribution of the differences is illustrated in Figure 27.

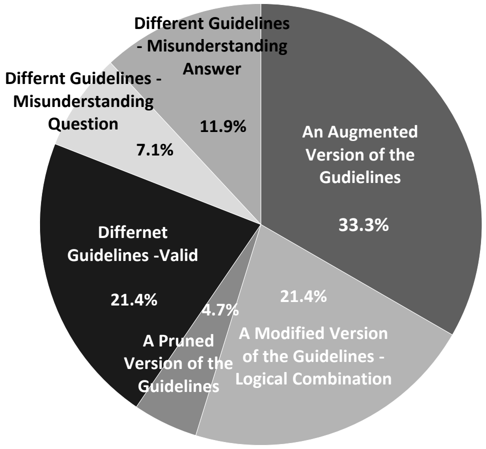


Figure 27. The Distribution of the Differences between the Recommendations Related to Acceptable-Alternative Ratings (N=38)

Overall, in more than half of the "*acceptable-alternative*" ratings (i.e., 59.1%), the evaluators had either augmented, modified, or pruned the recommendations of RAPID. The modifications were either some API design mechanisms that were not available in the knowledge base of RAPID, or case-specific details which were sometimes related to the design of back-end systems rather than the API. During studying these suggestions, we encountered 7 new API design mechanisms to address some non-functional requirements that were not available in RAPID's knowledge base. Out of the nine comments categorized as modifying the logical relationships, six of them were related to altering "OR" relationship between some design techniques to "XOR" in the rule base of RAPID, due to limitations in the implementation of the recommendation component. Moreover, in 21.4% of the cases, the evaluators had suggested a completely guideline different from RAPID. Some of these different guidelines were not available in the knowledge base of RAPID while others were referring to a specific technology or implementation rather than a mechanism for design. We specifically encountered 6 new design guidelines that were not available in RAPID's knowledge base. Finally, in 19% of the cases, some evaluators had misunderstood the design question or the provided answers hinting that the design and communication of the exam notebook had affected the ratings of the evaluators.

While analyzing "*acceptable-alternative*" ratings, we also noticed that there are some overlaps between some "*acceptable-alternative*" and "*unacceptable – the answer is partly incomplete or invalid*." ratings, specifically where more design guidelines were required to answer a question. In similar cases, where the answer had to be augmented with more guidelines to answer a specific question, some evaluators had considered an answer as "*acceptable-alternative*" while some others had considered the same answer as "*unacceptable – partly incomplete or irrelevant.*"

The above insight suggests that statistically the main differences between the recommendations of the domain experts and the recommendations of RAPID were (1) the addition of some design guidelines (33.3%), (2) modification of the logical combination of the design guidelines (21.4%) (most of which were related to altering "XOR" relationships to "OR" due to limitations in the implementation of RAPID), and (3) suggesting completely different guidelines (21.4%).

As a final point, while analyzing the evaluators' comments, we noticed that only for two out of the thirty questions, the majority of the experts had suggested the same answer. In other cases, the design recommendation of the experts varied and there was no agreement among the suggested answers.

7 Related Work

Numerous research projects have been devoted to building tools that assist with software development activities. Many of these projects focus on programming and aim to facilitate the task of writing code via automating program generation or via recommending code snippets [40, 41, 42]. The main idea behind these projects is to automatically generate code from user intent expressed in natural language. Some of the developed tools and prototypes include the PSI Program Synthesizer (1976) [43], the Programmer's Apprentice (1988) [44], Strathcona (2005) [45, 46], Prospector (2005) [47], ParseWeb (2007) [48], MAPO (2009) [49], Code Suggestion Plugin for Eclipse (2012) [50], SLANG Code Completer (2014) [51], Bing Developer Assistant (2015) [52, 53], A Programmer's Apprentice Again (2015) [54], and Amanuensis (2018) [55]. One common approach to develop these tools is to re-use large corpora of already written code available either on the web (such as public code repositories, open-source projects, code search engines, and developers' community platforms), or in the local repositories, libraries, and APIs of a programming language. Some of the built tools (such as Strathcona [45], Prospector [47], ParseWeb [48], MAPO [49], and Bing Developer Assistant [52, 53]) search and mine the available corpora to find code fragments relevant to a programmer's query, rank the results and return the top answers to the programmer. Others (such as Code Suggestion Plugin for Eclipse [50] and SLANG code completer [51]) use the available code corpora for statistical purposes and suggest code words and lines which frequently appear together. In parallel to these efforts, a few other research efforts (such as Draco [56]) formalize and encode design knowledge, and use the encoded knowledge to synthesize and recommend code fragments for a given query.

The approach adopted in developing RAPID is complementary to the above research efforts. First, addressing non-functional in the design of software systems is language-independent. For example, mechanisms adopted to address requirements such as access confidentiality of APIs are rather generic and once selected can be linked to a code fragment in a language. Second, selection from among these design techniques requires the knowledge of trade-offs supported by empirical evidence or expert analysis. This kind of knowledge cannot be found in existing code corpora written by developers who themselves may have various levels of expertise and knowledge about the design and implementation of a specific requirement. The knowledge about the range of techniques to address a specific non-functional and their trade-offs is available in different corpora, such as books and scientific articles. These corpora should also be utilized in parallel to code corpora to recommend appropriate techniques to developers. In this paper, we explored one possible way to utilize various design knowledge corpora to assist with designing APIs.

Knowledge-based software assistance is a long-lasting idea in the field of software engineering [57] and particularly in the sub-field of software architecture. Various research studies emphasize the need for reusing architectural knowledge, including [58, 59]. Moreover, several rational methodologies and concepts have been proposed (e.g., Architecture Trade-Offs Analysis Method (ATAM) [60], Architecture Rationale and Element Linkage (AREL) [61], and Trade-off-oriented Development (ToF) [62]) that considers the knowledge of various design alternatives and their trade-offs in making architectural design decisions. The approach adopted to develop RAPID connects the idea of reusing architectural design knowledge to rational design and provides a concrete realization of these concepts.

RAPID employs a semi-formal and step-wise inference procedure over a specific multi-valued logic to transform given requirements specifications into design fragments. There are numerous formal frameworks achieving the same objective, including [65]. However, these frameworks mainly focus on the functional and structural design of software systems and do not address non-functional requirements.

In RAPID, the NFR multi-valued logic is used to encode and reason about the collected API design knowledge. Already, there are tools, including the CGM Tool [66], implementing the NFR logic or extended interpretations of it. However, these tools are mainly designed to support graphical requirements modeling and reasoning in interaction with a user. They do not immediately suite the iterative and automated reasoning steps as required for our research. We thus decided to implement the portion of the NFR logic that was required for RAPID from scratch. From-scratch development may raise concerns about correct implementation and operation of the tool. To address these concerns, we have made the source code of the tool available for further scrutiny.

The step related to formalizing software design knowledge has been previously attempted in [63, 64]. These efforts develop goal graphs from design knowledge available in the form of design patterns. However, they lack a systematic procedure for encoding knowledge. To develop RAPID, we have devised a systematic procedure for structuring and formalizing design knowledge. The adopted approach aims to reduce the need to human interpretation during knowledge encoding, and to increase the validity and reproducibility of the encoded knowledge.

8 Discussion and Conclusions

Although dealing with non-functional requirements during software design has been long studied, there is still no framework to particularly support software developers with addressing these requirements in designing web application programming interfaces. To address this gap, we have developed and evaluated (Rational API Designer) RAPID, a knowledge-based assistant that aids software developers in addressing non-functional requirements in the design of web APIs. RAPID assists its users via elaborating non-functional and functional requirements of a design case and recommending web API design fragments that meet the expected requirements. To suggest design guidelines, RAPID considers the trade-offs of design fragments against a variety of non-functional requirements.

Currently, RAPID's knowledge base contains 115 distinct terms and 156 implication rules encoding a body of knowledge about designing Web API. Out of the available 115 distinct terms, 35 of them are non-functional requirements, and 80 of them are functional design objectives and techniques. Moreover, RAPID provides design guidelines for refining and designing 7 groups of non-functional requirements, refining and designing 24 functional design objectives, and analyzing the trade-offs of 22 distinct design techniques.

The evaluation study reported in this paper allowed us to evaluate the accuracy of RAPID in answering thirty design questions for seven cases of web API design collected from the API specifications of various open software platforms available on the Web. To evaluate the design recommendations, we asked seven well-experienced domain experts to assess the answers and considered the aggregate opinion of the experts in recognizing an answer as valid and acceptable. The results of the evaluation show that in 73.3% of the times (i.e., in 22 out 30 answers), the answers of RAPID were considered as acceptable by the majority of the seven domain experts.

Analysis of the evaluators' comments on the 8 answers that were not accepted by the majority of the evaluators suggests that two problems have played a role in the unacceptance of these answers: (1) incompleteness of the design guidelines while they were valid, and (2) misunderstanding or incomplete understanding of some answers or design questions by the evaluators.

Moreover, the overall analysis of the sixty unacceptable ratings (i.e., "*partly unacceptable*", and "*completely unacceptable*") given by the domain experts to some answers suggests that more than one third of these ratings (33.8%) is due to the incompleteness of the answers. This incompleteness is partly due to lack of enough API design knowledge and in part due to lack of case-specific details in the answers. Moreover, 38.2% of the unacceptable ratings given to the answers is due to some misunderstanding of the design questions or answers, hinting to two potential issues: (1) the short answers that RAPID currently provides without explaining the recommended guidelines, and (2) the shortcomings in the design of the API exam notebook and its communication with the domain experts.

Additionally, the overall analysis of the thirty-eight "*acceptable-alternative*" ratings supported by the evaluators' comments suggests that in more than half of the cases (i.e., 59.1 %), the suggestion of the evaluators was a modification of RAPID's recommendation. Only in 21% cases, the evaluators' recommendations completely differed from the guidelines that RAPID had provided. Moreover, like the unacceptable ratings, in 19% of the "*acceptable-alternative*" ratings, evaluators had differently or incompletely understood the design question or the answer, again hinting to the lack of explanation capabilities in RAPID and the problems in the communication of the API design exam notebook.

Finally, out of all the evaluators' comments supporting the unacceptable and "*acceptable-alternative*" ratings, thirteen of them were about the logical relationships between the recommended design guidelines. Out of these thirteen comments, seven of them were related to the answers in which the "OR" relationships between design techniques had been altered to "XOR" due to limitations in the implementation of the reasoning component of RAPID.

Based on the above insight, we expect that extending and refining the API design knowledge, adding explanation capabilities, enriching the reasoning and recommendation components of RAPID, and improving the design and communication of the API design exam with domain experts will improve the obtained accuracy measure.

Currently, RAPID has been operationalized and evaluated as a working prototype within a small research setting. To prepare RAPID to function within a field setting, several future steps need to be performed:

- *Enriching and further evaluating RAPID's knowledge of API design:* Currently, the knowledge of RAPID is limited to the 156 rules encoding an initial body of web API design knowledge. These rules have been developed in three consecutive research steps: (1) extracting and aggerating a body of web API design knowledge from various knowledge sources, (2) summarizing the structured knowledge using a set of knowledge graphs, and (3) translating the summarized knowledge into a specific multi-valued logic. In the evaluation study, we assumed that the structured body of knowledge, the knowledge graphs, and the related rules are valid by construction as a systematic approach has been adopted to develop them. However, to ensure the verity of the design knowledge, the outcome of each step needs to be separately examined by a group of domain experts. Moreover, the collected body of API design knowledge needs to be further extended and refined. Based on the insight obtained from the evaluation study, we expect as RAPID's knowledge of API design is extended, its performance in providing accurate and reliable design consultations will increase.
- *Further evaluation of RAPID's accuracy:* RAPID has been tested with a few cases of API design prepared based on real-world web API specifications. Further evaluation studies are required to assess the validity and reliability of RAPID's design recommendations. RAPID needs to be tested with a wide variety of API design specifications directly collected from the field, specifying the requirements of prospective or retrospective web API design cases, and preferably by researchers who have not been involved in the development of RAPID. Moreover, design recommendations of RAPID have been reviewed and evaluated by a group of seven domain experts. To achieve a higher level of confidence about the accuracy of the generated design guidelines, a larger group of evaluators including ten to twelve domain experts need to be used in future evaluation studies.
- *Enriching the explanation capabilities of RAPID:* RAPID performs a step-wise inference procedure to arrive from a given non-functional or related functional requirement to a final design, and it shows each step to the user. Adding explanation capabilities to RAPID, including defining the requirements and describing the design techniques will enrich the power of RAPID as a consultant, and will remove possible ambiguities that a user may encounter in interacting with RAPID. In the evaluation study, we observed cases where the domain experts had misunderstood the answers or their relation to the design questions. We thus expect that the explanation of design guidelines will improve the quality of evaluation by domain experts and will affect the obtained accuracy measure.
- *Enriching the reasoning and recommendation capabilities of RAPID:* Currently, the recommendation component of RAPID only supports "XOR" relationships for selecting from among alternative design techniques. Due to this limitation, some "OR" relationships between the design technique entities in the knowledge domain have been altered to "XOR" in RAPID's rule base. To increase the accuracy of design recommendations, reasoning about "*OR*" logical relationships should be added to RAPID. Moreover, currently, the reasoning component of RAPID implements a forward chaining procedure to evaluate the statements expressed in the NFR logic. Using backward chaining procedures will improve the time-complexity of the inference procedure.
- *Testing the effectiveness and usability of RAPID as a design assistant:* We expect that using RAPID will increase the efficiency of developers in addressing non-functional or functional requirements in APIs. Controlled experiments are required to compare the impact of RAPID on the time-to-completion of answering design questions against performing an ad-hoc research on the Web. Moreover, RAPID should be easy-to-use to be adopted by

developers. To obtain insight about how to improve the interactions of RAPID with users, user and case studies need to be performed investigating the experience of various users in real-world scenarios.

- *Reducing the involvement of researchers in collecting, extracting, and encoding design knowledge*: Structuring and encoding design knowledge is a time-consuming procedure. Automating these steps will facilitate the extension and refinement of RAPID's design knowledge. Further studies are required to explore automatic extraction, aggregation, summarization, and encoding of design knowledge.
- *Extending the scope of RAPID's design assistance*: The assistance that RAPID provides is currently limited to the domain of web API design. To generalize the scope of the assistance, further studies are required to investigate whether the performed steps to extract, encode and reason about web API design knowledge can be applied in other design domains, and whether it is possible to populate and use RAPID in other design domains.
- *Reducing the involvement of user in reaching from a given topic to a final answer:* Currently, a user is fully involved in interactions with RAPID to reach from a specified requirement to a piece of design guideline. Further studies are required to explore how to minimize the role of a user and how to fully automate the generation of design recommendations for a given requirements specification.

9 Acknowledgement

The first author thanks 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 the architects and software engineers at Google and Autodesk, and the domain experts at the University of Toronto for evaluating the results of this research and providing invaluable feedback. For the sake of anonymity and conformance with research ethics, we only mention their first names: Andy, Clayton, Daniel, Nick, Patrick, Rami, Saeed, and Zia.

10 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.
2. Sadi, M. H., & Yu, E. (2014). Analyzing the evolution of software development: From creative chaos to software ecosystems. In *2014 IEEE 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. Weber, R. H. (2010). Internet of Things–New security and privacy challenges. Computer law & security review, 26(1), 23-30.
6. 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.
7. Myers, B. A., & Stylos, J. (2016). Improving API usability. Communications of the ACM, 59(6), 62-69.
8. Siriwardena, P. (2014). Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE. Apress, Berkeley, CA.
9. De, B. (2017). API Management: An Architect's Guide to Developing and Managing APIs for Your Organization. Apress, Berkeley, CA, First edition March 2017.
10. Vijayakumar, T. (2018). Practical API Architecture and Development with Azure and AWS. Apress, Berkeley, CA.
11. Akamai White Paper 1, Strategies for API protection, Available at https://www.akamai.com/us/en/multimedia/documents/white-paper/akamai-strategies-for-api-security-white-aper.pdf?utm_source=twitter&utm_medium=social_organic&utm_campaign=api

12. Akamai White Paper 2, Solving API Performance, Reliability, & Security Challenges, Available at https://www.akamai.com/uk/en/multimedia/documents/product-brief/akamai-solving-api-security-performance-for-enterprise-applications-focus-sheet.pdf

13. RFC 6749 (2012) - The OAuth 2.0 Authorization Framework, Available at https://tools.ietf.org/html/rfc6749 on October 2012.

14. Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., & Mortimore, C. (2014). OpenID Connect Core 1.0 incorporating errata set 1. The OpenID Foundation, specification.

15. Sakimura, N., Bradley, D., de Mederiso, B., Jones, M., & Jay, E. (2012). OpenID connect standard 1.0-draft 09.

16. 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.

17. 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.

18. 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.

19. 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.

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. Richardson, C. (2015). Pattern: API Gateway. Backend for Front-End, 37-40, available at http://microservices.io/patterns/api-gateway.html.

23. Kitchenham, B. (2004). Procedures for performing systematic reviews. Keele, UK, Keele University, 33(2004), 1-26.

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

25. 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 (p. 38). ACM.

26. 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.

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

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

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

30. 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.

31. 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). Springer, Berlin, Heidelberg.

32. Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., ... & Navigli, R. (2020). Knowledge graphs. *arXiv preprint arXiv:2003.02320*.

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

34. Susskind, L. E., McKearnen, S., & Thomas-Lamar, J. (1999). *The consensus building handbook: A comprehensive guide to reaching agreement*. Sage Publications.

35. Jarke, M., Loucopoulos, P., Lyytinen, K., Mylopoulos, J., & Robinson, W. (2011). The brave new world of design requirements. *Information Systems*, *36*(7), 992-1008.requirements. Information Systems, 36(7), 992-1008.

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

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

38. Saldaña, J. (2009). The coding manual for qualitative researchers. Sage.

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

40. Gulwani, S., Polozov, O., & Singh, R. (2017). Program synthesis. Foundations and Trends® in Programming Languages, 4(1-2), 1-119.

41. Alur, R., Singh, R., Fisman, D., & Solar-Lezama, A. (2018). Search-based program synthesis. Communications of the ACM, 61(12), 84-93.

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

43. Green, C. (1976). The design of the PSI program synthesis system. In Proceedings of the 2nd international conference on Software engineering (pp. 4-18). IEEE Computer Society Press.

44. Rich, C., & Waters, R. C. (1988). The Programmer's Apprentice: A research overview. Computer, 21(11), 10-25.

45. Holmes, R., & Murphy, G. C. (2005). Using structural context to recommend source code examples. In Proceedings of the 27th international conference on Software engineering (pp. 117-125).

46. Holmes, R., Walker, R. J., & Murphy, G. C. (2005). Strathcona example recommendation tool. In Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (pp. 237-240).

47. Mandelin, D., Xu, L., Bodík, R., & Kimelman, D. (2005). Jungloid mining: helping to navigate the API jungle. ACM Sigplan Notices, 40(6), 48-61.

48. Thummalapenta, S., & Xie, T. (2007). Parseweb: a programmer assistant for reusing open source code on the web. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (pp. 204-213).

49. Zhong, H., Xie, T., Zhang, L., Pei, J., & Mei, H. (2009). MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming* (pp. 318-343). Springer, Berlin, Heidelberg.

50. Hindle, A., Barr, E. T., Su, Z., Gabel, M., & Devanbu, P. (2012). On the naturalness of software. In 2012 34th International Conference on Software Engineering (ICSE) (pp. 837-847). IEEE.

51. Raychev, V., Vechev, M., & Yahav, E. (2014). Code completion with statistical language models. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (pp. 419-428).

52. Zhang, H., Jain, A., Khandelwal, G., Kaushik, C., Ge, S., & Hu, W. (2016). Bing developer assistant: improving developer productivity by recommending sample code. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foun-dations of Software Engineering (pp. 956-961).

53. Wei, Y., Chandrasekaran, N., Gulwani, S., & Hamadi, Y. (2015). Building Bing developer assistant. Technical Report. MSR-TR-2015-36, Microsoft Research.

54. Shrobe, H., Katz, B., & Davis, R. (2015). *Towards a programmer's apprentice (again)*. Center for Brains, Minds and Machines (CBMM).

55. Dean, T., Chiang, M., Gomez, M., Gruver, N., Hindy, Y., Lam, M., ... & Wang, L. (2018). Amanuensis: The Programmer's Apprentice. *arXiv preprint arXiv:1807.00082*.

56. Moritz, D., Wang, C., Nelson, G. L., Lin, H., Smith, A. M., Howe, B., & Heer, J. (2019). Formalizing visualization design knowledge as constraints: Actionable and extensible models in Draco. IEEE transactions on visualization and computer graphics, 25(1), 438-448.

57. Green, C., Luckham, D., Balzer, R., Cheatham, T., & Rich, C. (1986). Kestrel Institute: Report on Knowledge-Based Software Assistant. In Readings in Artificial Intelligence and Software Engineering (pp. 377-428). Morgan Kaufmann.

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

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

60. 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.

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

62. 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 (ICSE-NIER) (pp. 109-112). IEEE.

63. Sadi, M. H., & Yu, E. (2017a). 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.

64. Sadi, M. H., & Yu, E. (2017b). Accommodating Openness Requirements in Software Platforms: A Goal-Oriented Approach. In International Conference on Advanced Information Systems Engineering (pp. 44-59). Springer, Cham.

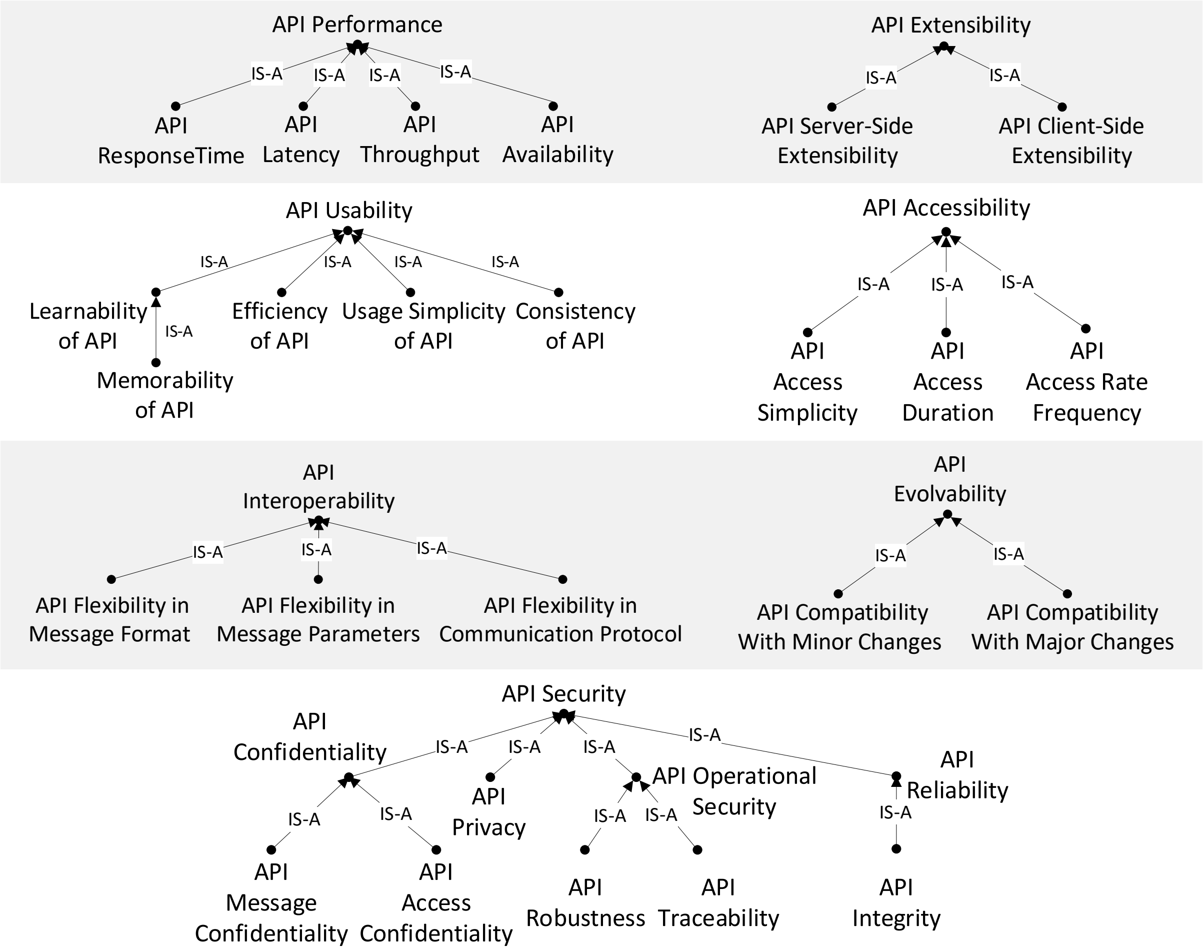
65. Jackson, D. (2012). Software Abstractions: logic, language, and analysis. MIT press.

66. Nguyen, M. C., Sebastiani, R., Giorgini, P., and Mylopoulos, J. (2018). Multi-Objective Reasoning with Constraint Goal Models Requirements Engineering. Requirements Engineering, 23(2), 189-225.

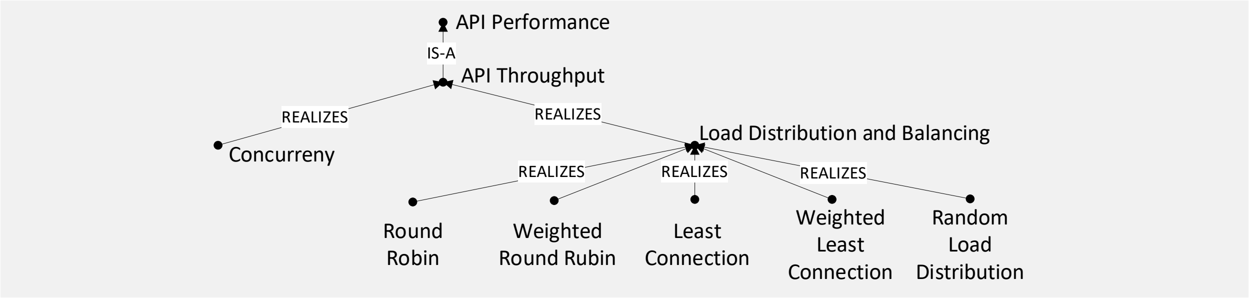
Appendices

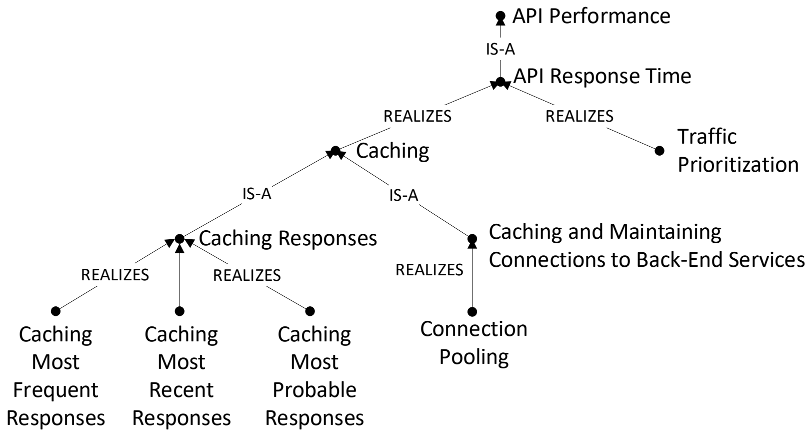
Appendix 1: The API Design Knowledge Graphs

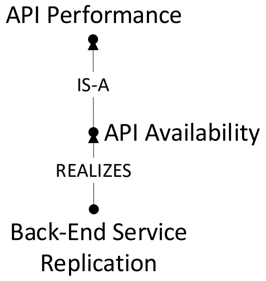
1. The API Non-Functional Requirements

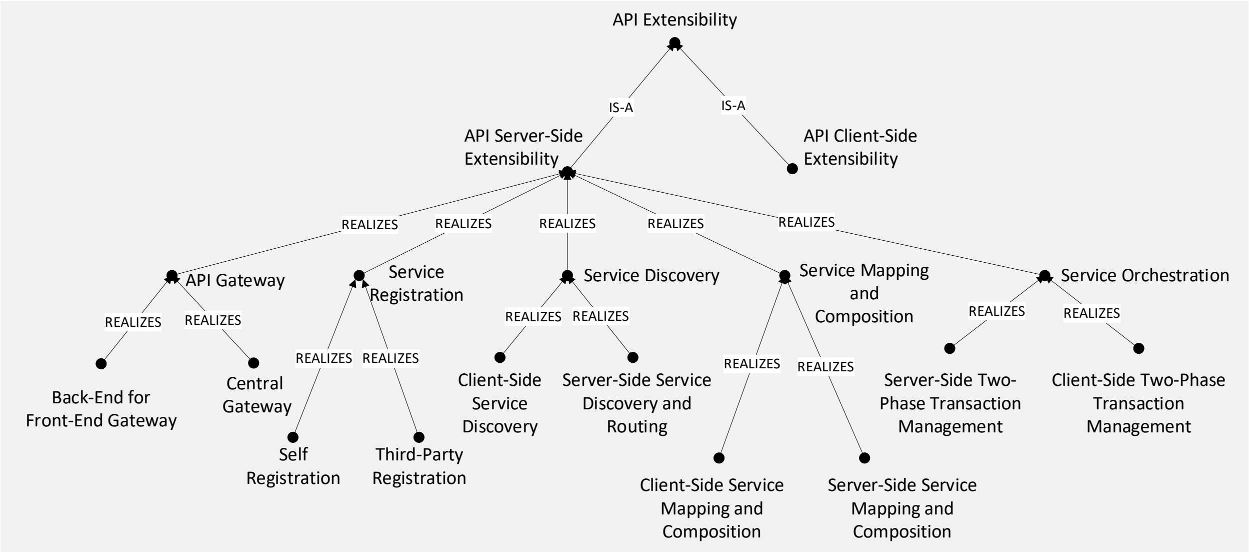


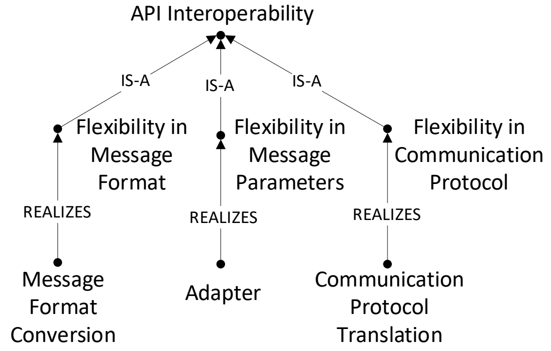
2. The API Design Techniques

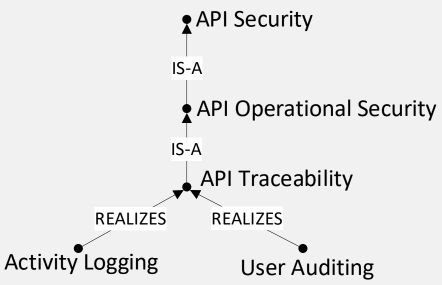


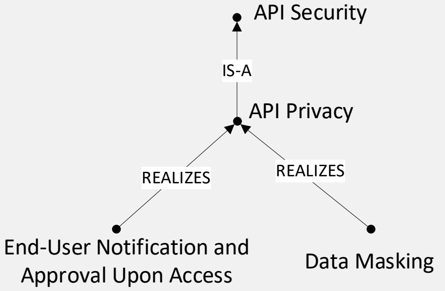

API Performance
IS-A
API Response Time
REALIZES
REALIZES
Caching
Traffic Prioritization
IS-A
IS-A
Caching Responses
Caching and Maintaining Connections to Back-End Services
REALIZES
REALIZES
REALIZES
Caching Most Frequent Responses
Caching Most Recent Responses
Caching Most Probable Responses
Connection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

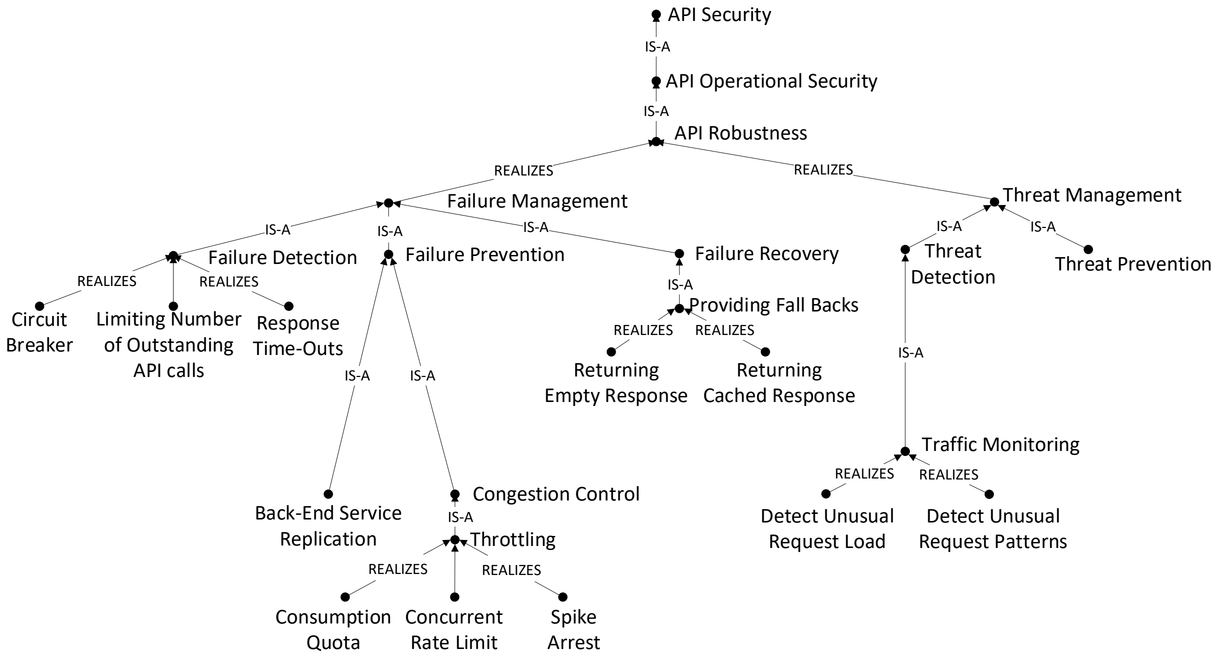

API Performance
IS-A
API Availability
REALIZES
Back-End Service Replication

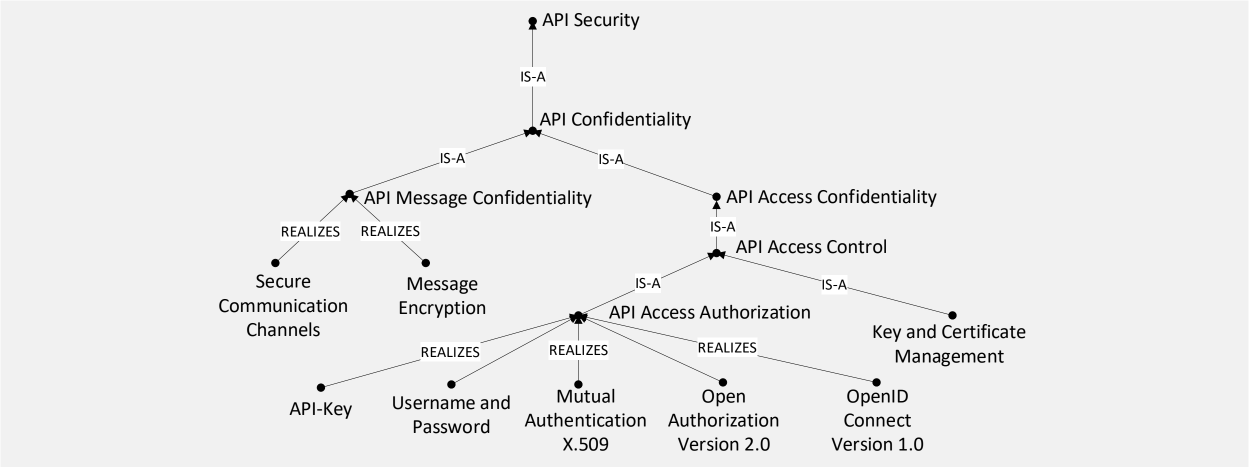

API Extensibility
IS-A
IS-A
API Server-Side Extensibility
API Client-Side Extensibility
REALIZES
REALIZES
REALIZES
REALIZES
REALIZES
API Gateway
Service Registration
Service Discovery
Service Mapping and Composition
Service Orchestration
REALIZES
REALIZES
Back-End for Front-End Gateway
Central Gateway
REALIZES
REALIZES
Self Registration
Third-Party Registration
REALIZES
REALIZES
Client-Side Service Discovery
Server-Side Service Discovery and Routing
REALIZES
REALIZES
Client-Side Service Mapping and Composition
Server-Side Service Mapping and Composition
REALIZES
REALIZES
Server-Side Two-Phase Transaction Management
Client-Side Two-Phase Transaction Management

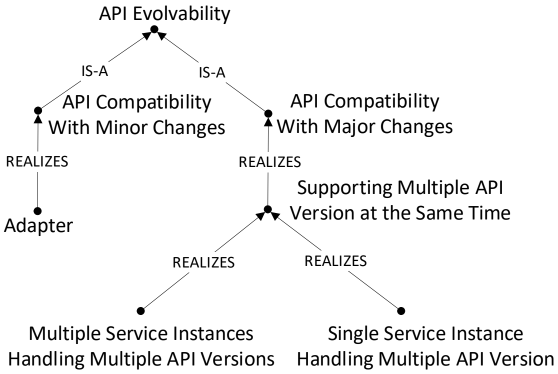

API Interoperability
IS-A
IS-A
IS-A
Flexibility in Message Format
Flexibility in Message Parameters
Flexibility in Communication Protocol
REALIZES
REALIZES
REALIZES
Message Format Conversion
Adapter
Communication Protocol Translation

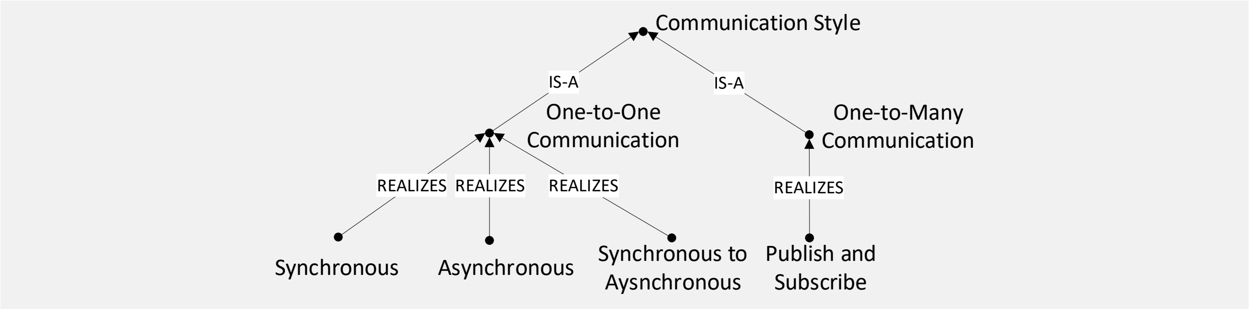

API Security
IS-A
API Operational Security
IS-A
API Traceability
REALIZES
REALIZES
Activity Logging
User Auditing


API Security
IS-A
API Privacy
REALIZES
REALIZES
End-User Notification and Approval Upon Access
Data Masking

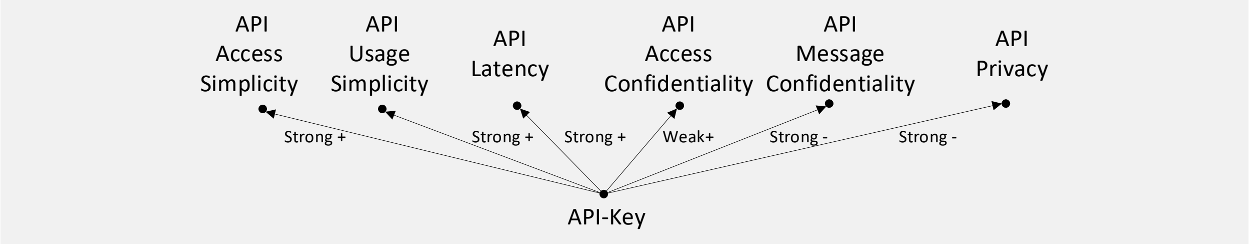

API Security
IS-A
API Operational Security
IS-A
API Robustness
REALIZES
REALIZES
Failure Management
Threat Management
IS-A
IS-A
IS-A
IS-A
IS-A
Failure Detection
Failure Prevention
Failure Recovery
Threat Detection
Threat Prevention
REALIZES
REALIZES
Circuit Breaker
Limiting Number of Outstanding API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

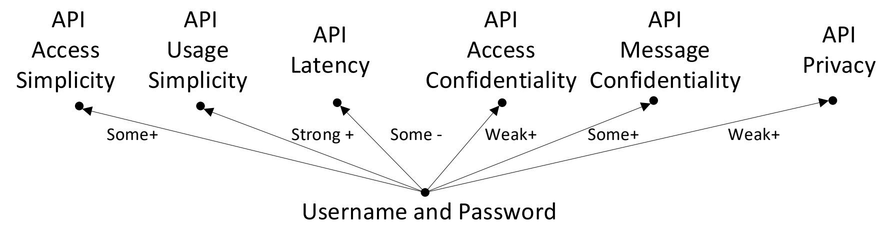

API Security
IS-A
API Confidentiality
IS-A
IS-A
API Message Confidentiality
API Access Confidentiality
REALIZES
REALIZES
Secure Communication Channels
Message Encryption
IS-A
API Access Control
IS-A
IS-A
API Access Authorization
Key and Certificate Management
REALIZES
REALIZES
REALIZES
API-Key
Username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

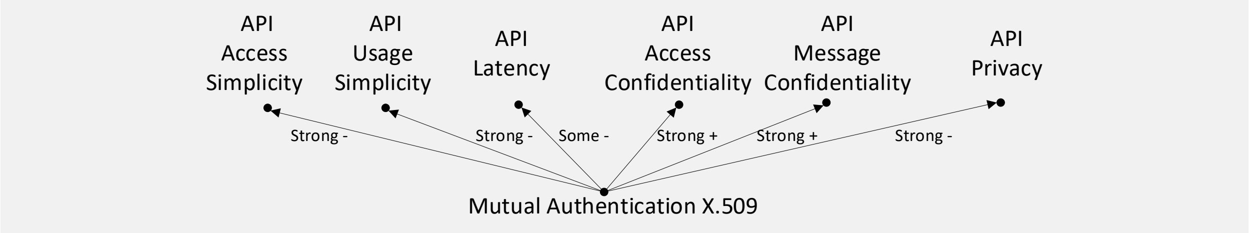

API Evolvability
IS-A
IS-A
API Compatibility With Minor Changes
API Compatibility With Major Changes
REALIZES
REALIZES
Adapter
Supporting Multiple API Version at the Same Time
REALIZES
REALIZES
Multiple Service Instances Handling Multiple API Versions
Single Service Instance Handling Multiple API Version

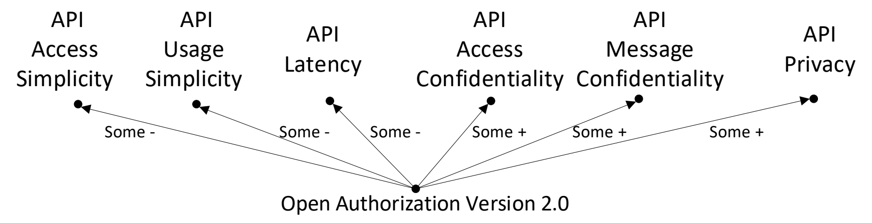


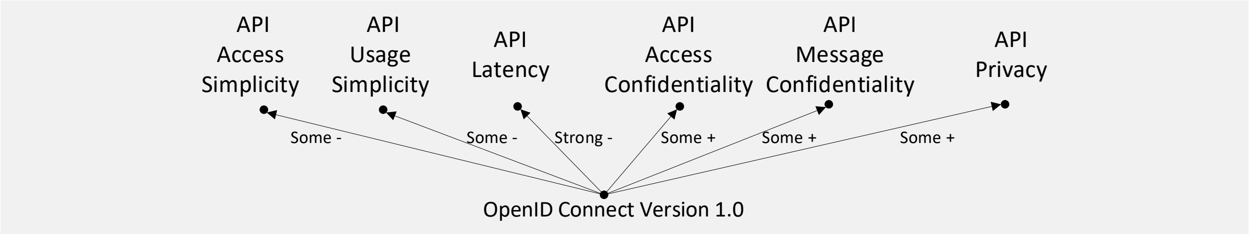
3. Trade-Offs of the API Design Techniques

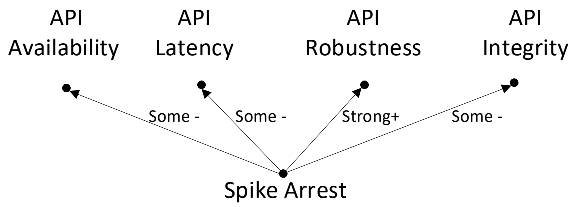


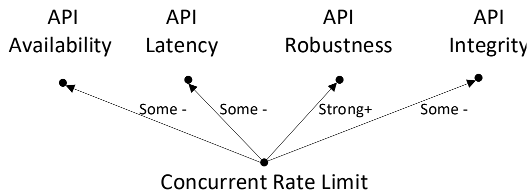


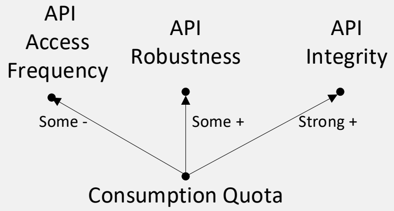


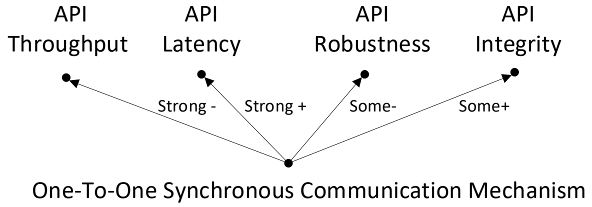


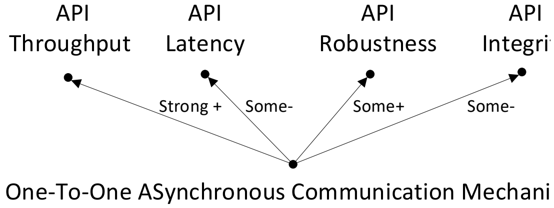


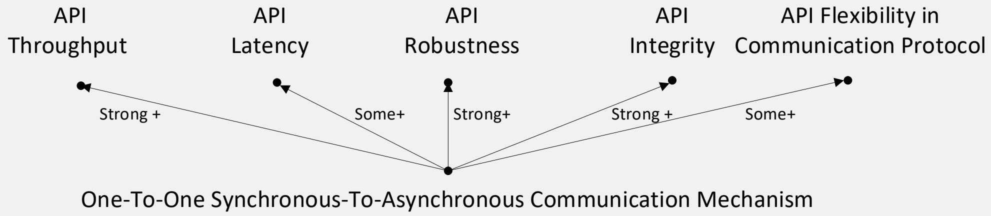

API Availability
API Latency
API Robustness
API Integrity
Some -
Some -
Strong+
Some -
Spike Arrest

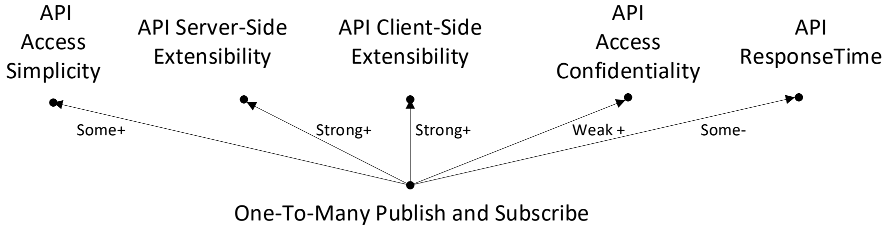

API Availability
API Latency
API Robustness
API Integrity
Some -
Some -
Strong+
Some -
Concurrent Rate Limit

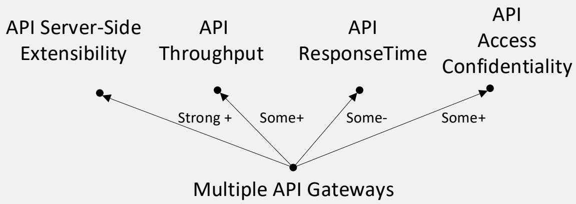

API Access Frequency
API Robustness
API Integrity
Some -
Some +
Strong +
Consumption Quota

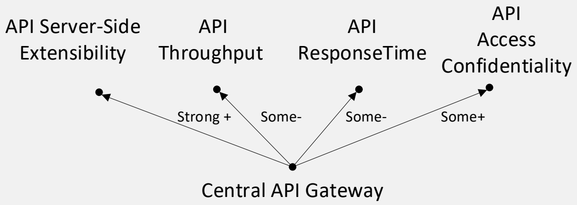

API Throughput
API Latency
API Robustness
API Integrity
Strong -
Strong +
Some-
Some+
One-To-One Synchronous Communication Mechanism

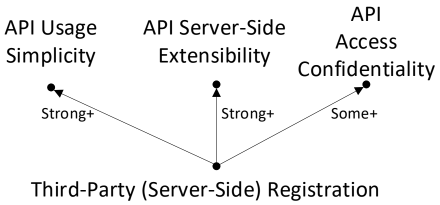

API Throughput
API Latency
API Robustness
API Integri
Strong +
Some-
Some+
Some-
One-To-One ASynchronous Communication Mechani

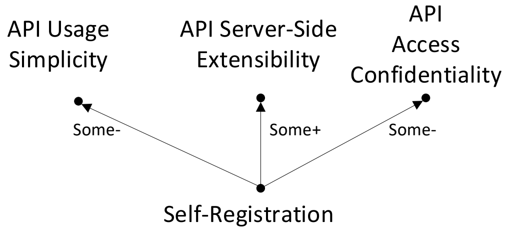

API Throughput
API Latency
API Robustness
API Integrity
API Flexibility in Communication Protocol
Strong +
Some+
Strong+
Strong +
Some+
One-To-One Synchronous-To-Asynchronous Communication Mechanism

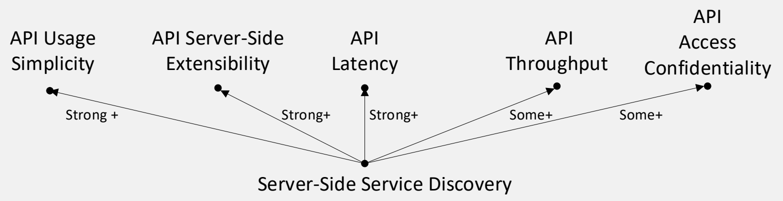

API Access Simplicity
API Server-Side Extensibility
API Client-Side Extensibility
API Access Confidentiality
API ResponseTime
Some+
Strong+
Strong+
Weak +
Some-
One-To-Many Publish and Subscribe

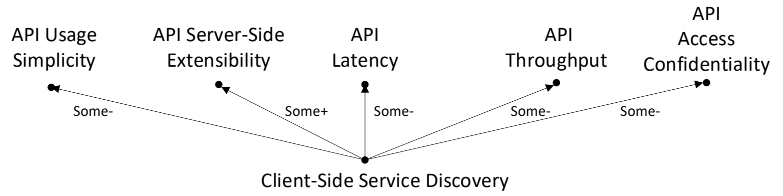

API Server-Side Extensibility
API Throughput
API ResponseTime
API Access Confidentiality
Strong +
Some+
Some-
Some+
Multiple API Gateways

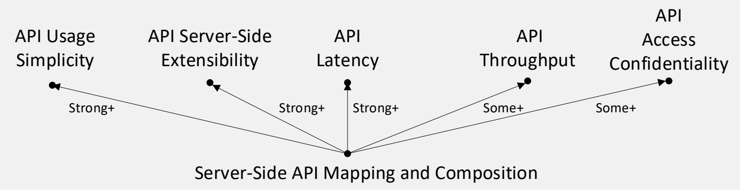

API Server-Side Extensibility
API Throughput
API ResponseTime
API Access Confidentiality
Strong +
Some-
Some-
Some+
Central API Gateway

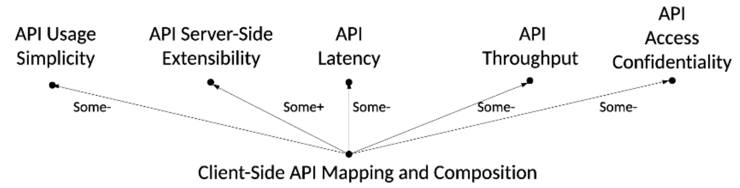

API Usage Simplicity
API Server-Side Extensibility
API Access Confidentiality
Strong+
Strong+
Some+
Third-Party (Server-Side) Registration

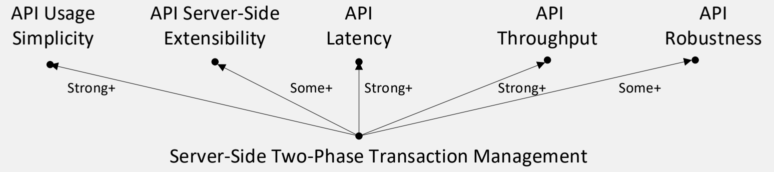

API Usage Simplicity
API Server-Side Extensibility
API Access Confidentiality
Some-
Some+
Some-
Self-Registration

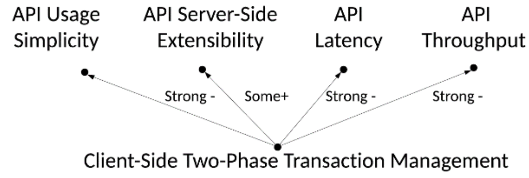

API Usage Simplicity
API Server-Side Extensibility
API Latency
API Throughput
API Access Confidentiality
Strong +
Strong+
Strong+
Some+
Some+
Server-Side Service Discovery


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


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


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


API Usage Simplicity
API Server-Side Extensibility
API Latency
API Throughput
API Robustness
Strong+
Some+
Strong+
Strong+
Some+
Server-Side Two-Phase Transaction Management


API Usage Simplicity
API Server-Side Extensibility
API Latency
API Throughput
Strong -
Some+
Strong -
Strong -
Client-Side Two-Phase Transaction Management

Appendix 2: A Brief Overview of NFR Multi-Valued Logic

1. Variables and Connectives

In NFR, variables are of the form $G_1, G_2, \ldots G_n$ and are referred to as soft goals. The relationship between variables are of type implication. The implication relationships are either positive: $G_i \xrightarrow{Help\ (+)} G_j$, $G_i \xrightarrow{Some+} G_j$, $G_i \xrightarrow{Make(++)} G_j$, or negative*:* $G_i \xrightarrow{Hurt\ (-)} G_j$, $G_i \xrightarrow{Some-} G_j$, $G_i \xrightarrow{Break(--)} G_j$, or unknown*:* $G_i \xrightarrow{Unknwon(?)} G_j$, or combinatorial $(G_1, \ldots, G_m) \xrightarrow{and} G_n$, $(G_1, \ldots, G_m) \xrightarrow{or} G_n$.

2. Evaluating Variables

Value Assignment

A soft goal G_i receives a satisfaction value which is in the range of {*Denied, Weakly Denied (or partially Denied), Conflict, Undetermined, Weakly Satisfied (or Partially Satisficed), Satisficed}.*

$$Sat\ (G_i)\ \in \{Den, Pden, U, Conf, PSat, Sat\}$$

The partial ordering between values is as follows:

$$Den\ < PDen\ \leq\ U\ \approx\ Conf\ \leq\ PSat\ < Sat$$

We use := to assign a satisfaction value to a soft goal (e.g., $G_i := Den$ or $G_i := Sat$). A soft goal receives a satisfaction value either as direct input or by propagation rules.

Value Propagation Rules

Once the satisfaction value of a soft goal G_i is determined, the satisfaction values of the existing statements of which G_i is a part can be checked using the propagation rules as described in the following.

$$Gi \xrightarrow{Help\ (+)} Gj\!:\ \left\{\begin{array}{ll} G_j := PSat & if\ G_i := Sat\ or\ G_i := PSat \\ G_j := PDen & if\ G_i := Den\ or\ G_i := PDen \\ G_j := G_i & if\ G_i := Conf\ or\ G_i := U \end{array}\right\}$$

$$Gi \xrightarrow{Hurt\ (-)} Gj\!:\ \left\{\begin{array}{ll} G_j := PDen & if\ G_i := Sat\ or\ G_i := PSat \\ G_j := PSat & if\ G_i := Den\ or\ G_i := PDen \\ G_j := G_i & if\ G_i := Conf\ or\ G_i := U \end{array}\right\}$$

$$Gi \xrightarrow{Make\ (++)} Gj\!:\ G_j := G_k$$

$$Gi \xrightarrow{Break\ (--)} Gj\!:\ \left\{\begin{array}{ll} G_j := Den & ifG_i := Sat\ or\ G_i := PSat \\ G_j := PSat & if\ G_i := Den\ or\ G_i := PDen \\ G_j := G_i & if\ G_i = Conf\ or\ G_i := U \end{array}\right\}$$

$$Gi \xrightarrow{Some+} Gj\!:\ \left\{\begin{array}{ll} G_j := PSat & ifG_i := Sat\ or\ G_i := PSat \\ G_j := PDen & if\ G_i := Den\ or\ G_i := PDen \\ G_j := G_i & if\ G_i = Conf\ or\ G_i := U \end{array}\right\}$$

$$Gi \xrightarrow{Some-} Gj\!:\ \left\{\begin{array}{ll} G_j := PDen & ifG_i := Sat\ or\ G_i := PSat \\ G_j := PSat & if\ G_i := Den\ or\ G_i := PDen \\ G_j := G_i & if\ G_i = Conf\ or\ G_i := U \end{array}\right\}$$

$$Gi \xrightarrow{Unknown(?)} Gj\colon\ G_j := U$$

$$Gi \xleftrightarrow{Equal\ (=)} Gj\colon\ G_j := Gi$$

$$(Gi, Gj) \xrightarrow{and} Gk\colon\ G_k := \min(G_i, G_j)$$

$$(Gi, Gj) \xrightarrow{or} Gk\colon\ G_k := \max(G_i, G_j)$$

$$Gi \xrightarrow{*} Gj\colon\ G_j := U\ \ if\ \ G_i := U$$

$$Gi \xrightarrow{*} Gj\colon \left\{ \begin{matrix} G_j := U & if\ G_i \xrightarrow{Unknown(?)} G_j\ \ and\ \ G_i := Conf \\ G_j := Conf & if\ \ G_i := Conf \end{matrix} \right\}$$

Value Resolution Rules

After each step of value propagation, a soft goal G_i may receive several possibly conflicting or inconsistent values. To determine the final value of the soft goal G_i, a set of value resolution rules are used. Some value resolutions rules have only one possible outcome while other may have several outcomes. Those resolution rules with several outcomes require human decision to identify the final value. The value resolution rules are as follows:

$$\frac{G_i \xrightarrow{Make\ (++)} G_k, \quad G_j \xrightarrow{Make(++)} G_k, \quad G_i := Sat, \quad G_j := Sat}{\dfrac{G_k := Sat, \quad G_k := Sat}{G_k := Sat}}$$

$$\frac{G_i \xrightarrow{Make\ (++)} G_k, \quad G_j \xrightarrow{Break\ (--)} G_k, \quad G_i := Den, \quad G_j := Sat}{\dfrac{G_k := Den, \quad G_k := Den,}{G_k := Den}}$$

$$\frac{G_i \xrightarrow{Break\ (--)} G_k, \quad G_j \xrightarrow{Break(--)} G_k, \quad G_i := Sat, \quad G_j := Sat}{\dfrac{G_k := Den, \quad G_k := Den,}{G_k := Den}}$$

$$\frac{G_i \xrightarrow{Make\ (++)} G_k, \quad G_j \xrightarrow{Break\ (--)} G_k, \quad G_i := Sat, \quad G_j := Sat}{\dfrac{G_k := Sat, \quad G_k := Den}{G_k := Conf}}$$

$$\frac{G_i \xrightarrow{Help\ (+)} G_k, \quad G_j \xrightarrow{Help(+)} G_k, \quad G_i := Sat, \quad G_j := Den}{\dfrac{G_k := PSat, \quad G_k := PDen}{G_k := Conf}}$$

$$\frac{G_i \xrightarrow{Some\ +} G_k, \quad G_j \xrightarrow{Some+} G_k, \quad G_i := Sat, \quad G_j := Den}{\dfrac{G_k := PSat, \quad G_k := PDen}{G_k := Conf}}$$

$$\frac{G_i \xrightarrow{Hurt\ (-)} G_k, \quad G_j \xrightarrow{Hurt(-)} G_k, \quad G_i := Sat, \quad G_j := Den}{\dfrac{G_k := PDen, \quad G_k := PSat}{G_k := Conf}}$$

$$\dfrac{\dfrac{G_i \xrightarrow{Some-} G_k, \quad G_j \xrightarrow{Some-} G_k, \quad G_i := Sat, \quad G_j := Den}{G_k := PDen, \quad G_k := PSat}}{G_k := Conf}$$

$$\dfrac{\dfrac{G_i \xrightarrow{reak\ (--)} G_k, \quad G_j \xrightarrow{Break(--)} G_k, \quad G_i := Sat, \quad G_j := Den}{G_k := Den, \quad G_k := PSat}}{Human\ decision\ required -\ G_k := PDen\ or\ G_k := Den}$$

$$\dfrac{\dfrac{G_i \xrightarrow{Break(--)} G_k, \quad G_j \xrightarrow{Make(++)} G_k, \quad G_i := Den, \quad G_j := Sat}{G_k := PSat, \quad G_k := Sat}}{Human\ decision\ required -\ G_k := PSat\ or\ G_k := Sat}$$

$$\dfrac{\dfrac{G_i \xrightarrow{Make\ (++)} G_k, \quad G_j \xrightarrow{Some-} G_k, \quad G_i := Sat, \quad G_j := Sat}{G_k := Sat, \quad G_k := PDen}}{Human\ decision\ requierd -\ G_k := Conf\ or\ G_k := PSat\ or\ G_k := Sat}$$

$$\dfrac{\dfrac{G_i \xrightarrow{Make\ (++)} G_k, \quad G_j \xrightarrow{Some-} G_k, \quad G_i := Sat, \quad G_j := Den}{G_k := Sat, \quad G_k := PSat}}{Huamn\ decision\ required -\ G_k := PSat\ or\ G_k := PSat}$$

$$\dfrac{\dfrac{G_i \xrightarrow{Make\ (++)} G_k, \quad G_j \xrightarrow{Some+} G_k, \quad G_i := Sat, \quad G_j := Den}{G_k := Sat, \quad G_k := PDen}}{Human\ decision\ required - G_k := Conf\ or\ G_k := PSat\ or\ G_k := Sat}$$

$$\dfrac{\dfrac{G_i \xrightarrow{Break\ (--)} G_k, \quad G_j \xrightarrow{Some+} G_k, \quad G_i := Sat, \quad G_j := Sat}{G_k := Den, \quad G_k := PSat}}{Human\ decision\ required -\ G_k := Conf\ or\ G_k := PDen\ or\ G_k := Den}$$

$$\dfrac{\dfrac{G_i \xrightarrow{Break\ (--)} G_k, \quad G_j \xrightarrow{Some+} G_k, \quad G_i := Sat, \quad G_j := Den}{G_k := Den, \quad G_k := PDen}}{Human\ decision\ requierd -\ G_k := PDen\ or\ G_k := Den}$$

$$\dfrac{\dfrac{G_i \xrightarrow{Break\ (--)} G_k, \quad G_j \xrightarrow{Some-} G_k, \quad G_i := Sat, \quad G_j := Den}{G_k := Den, \quad G_k := PSat}}{Human\ decision\ required -\ G_k := Conf\ or\ G_k := PDen\ or\ G_k := Den}$$

$$\dfrac{\dfrac{G_i \xrightarrow{Help\ (+)} G_k, \quad G_j \xrightarrow{Help\ (+)} G_k, \quad G_i := Sat, \quad G_j := Sat}{G_k := PSat, \quad G_k := PSat}}{Human\ decision\ required -\ G_k := PSat\ or\ G_k := Sat}$$

$$\dfrac{\dfrac{G_i \xrightarrow{Some+} G_k, \quad G_j \xrightarrow{Some+} G_k, \quad G_i := Sat, \quad G_j := Sat}{G_k := PSat, \quad G_k := PSat}}{Human\ decision\ required -\ G_k := PSat\ or\ G_k := Sat}$$

$$\dfrac{\dfrac{G_i \xrightarrow{Hurt\ (-)} G_k, \quad G_j \xrightarrow{Hurt\ (-)} G_k, \quad G_i := Sat, \quad G_j := Sat}{G_k := PDen, \quad G_k := PDen}}{Human\ decision\ required -\ G_k := PDen\ or\ G_k := Den}$$

$$\dfrac{\dfrac{G_i \xrightarrow{Some-} G_k, \quad G_j \xrightarrow{Some-} G_k, \quad G_i := Sat, \quad G_j := Sat}{G_k := PDen, \quad G_k := PDen}}{Human\ decision\ required -\ G_k := PDen\ or\ G_k := Den}$$

Appendix 3: The List of Rules Available to RAPID

1. (Access Simplicity [API], Access Duration [API], Access Rate [API]) $\xrightarrow{\text{and}}$ Accessibility [API] : NF-REF
2. (Compatibility with Minor Changes [API], Compatibility with Major Changes [API]) $\xrightarrow{\text{and}}$ Evolvability [API] : NF-REF
3. (Understandability [API], Efficiency [API], Usage simplicity [API], Consistency [API]) $\xrightarrow{\text{and}}$ Usability [API]: NF-REF
4. (Memorability [API]) $\xrightarrow{\text{Help}}$ Usability [API] : NF-REF
5. (Response Time [API], Latency [API], Throughput [API], Availability [API]) $\xrightarrow{\text{and}}$ Performance [API] : NF-REF
6. (Server-Side Extensibility [API], Client-Side Extensibility [API]) $\xrightarrow{\text{and}}$ Extensibility [API] : NF-REF
7. (Flexibility in Message Format [API], Flexibility in Message Parameter [API], Flexibility in Communication Protocol [API]) $\xrightarrow{\text{and}}$ Interoperability [API] : NF-REF
8. (Confidentiality [API], Privacy [API], Operational Security [API], Reliability [API]) $\xrightarrow{\text{and}}$ Security [API] : NF-REF
9. (Message Confidentiality [API], Access Confidentiality [API]) $\xrightarrow{\text{and}}$ Confidentiality [API] : NF-REF
10. (Robustness [API], Traceability [API]) $\xrightarrow{\text{and}}$ Operational Security [API] : NF-REF
11. (Integrity [API]) $\xrightarrow{\text{Help}}$ Reliability [API] : NF-REF
12. (Usability [API]) $\xrightarrow{\text{and}}$ Adoptability [API] : NF-REF
13. (Interface Translation [ ]) $\xrightarrow{\text{Help}}$ Compatibility with Minor Changes [API] : NF-OP
14. (Adapter [ ]) $\xrightarrow{\text{Make}}$ Interface Translation [ ] : F-OP
15. (Supporting Multiple Versions of API at the Same Time [ ]) $\xrightarrow{\text{Help}}$ Compatibility with Major Changes [API] : NF-OP
16. (Multiple Service Instances Handling Multiple API versions [ ], Single Service Instance Handling Multiple API versions [ ]) $\xrightarrow{\text{xor}}$ Supporting Multiple Versions of API at the Same Time [ ] : F-OP
17. (Access Control [API]) $\xrightarrow{\text{Help}}$ Access Confidentiality [API] : NF-OP
18. (Access Authorization [API], Key and Certificate Management [ ]) $\xrightarrow{\text{and}}$ Access Control [API] : F-REF
19. (API-Key [ ], Username and Password [ ], Mutual Certificate-Based Authentication X.509 [ ], Open Authorization Version 2.0 [ ], Open-ID Connect Version 1.0 [ ]) $\xrightarrow{\text{(x)or}[5]}$ Access Authorization [API] : F-OP
20. (Securing Communication Channels [ ], Message Encryption [ ]) $\xrightarrow{\text{or}}$ Message Confidentiality [API] : NF-OP
21. (End-User Notification and Approval upon API Access [ ], Data Masking [ ]) $\xrightarrow{\text{or}}$ Privacy [API] : NF-OP
22. (Activity Logging [API], User Auditing [API]) $\xrightarrow{\text{or}}$ Traceability [API] : NF-REF
23. (Failure Management [API], Threat Management [API]) $\xrightarrow{\text{or}}$ Robustness [API] : NF-OP
24. (Failure Detection [API], Failure Prevention [API], Failure Recovery [API]) $\xrightarrow{\text{or}}$ Failure Management [API] : F-REF
25. (Threat Detection [API], Threat Prevention [API]) $\xrightarrow{\text{or}}$ Threat Management [API] : F-REF
26. (Circuit Breaker [ ], Response Time-Outs [ ]) $\xrightarrow{\text{(x) or}}$ Failure Detection [API] : F-OP
27. (Back-End Service Replication [ ], Congestion Control [API]) $\xrightarrow{\text{or}}$ Failure Prevention [API] : F-REF
28. (Throttling [ ]) $\xrightarrow{\text{Make}}$ Congestion Control [API] : F-REF

---

[5] “OR” rule types that have been altered to “XOR” in RAPID’s rule base are represented as “(x)or”.

29. (Consumption Quota [ ], Concurrent Rate Limit [ ], Spike Arrest [ ]) $\xrightarrow{\text{(x) or}}$ Congestion Control [API] : F-OP
30. (Providing Fall Backs [ ]) $\xrightarrow{\text{Make}}$ Failure Recovery [API] : F-REF
31. (Returning Empty Responses [ ], Returning Cached Responses [ ]) $\xrightarrow{\text{(x)or}}$ Providing Fall Backs [API] : F-OP
32. (Traffic Monitoring [API]) $\xrightarrow{\text{Make}}$ Threat Detection [API] : F-REF
33. (Detecting Unusual Request Loads [ ], Detecting Unusual Request Patterns [ ]) $\xrightarrow{\text{or}}$ Traffic Monitoring [API] : F-REF
34. (Back-End Service Concurrency [ ] , Load Balancing and Distribution [ ]) $\xrightarrow{\text{or}}$Throughput [API] : NF-OP
35. (Round-Rubin Load Distribution [ ], Weighted Round-Rubin Load Distribution [ ], Least Connection Load Distribution [ ], Weighted Least Connection Load Distribution [ ], Random Load Distribution [ ]) $\xrightarrow{\text{xor}}$ Load Balancing and Distribution [ ] : F-OP
36. (Caching [ ], Request Traffic Prioritization [ ]) $\xrightarrow{\text{or}}$ Response Time [API]: F-REF
37. (Caching Responses [API], Caching and Maintaining Connections to Back-End Services [API]) $\xrightarrow{\text{or}}$ Caching [ ] : F-REF
38. (Caching Most Frequent Responses [ ], Caching Most Recent Responses [ ], Caching Most Probable Responses [ ]) $\xrightarrow{\text{(x)or}}$ Caching Responses [ ] : F-OP
39. (Connection Pooling [ ]) $\xrightarrow{\text{Make}}$ Caching and Maintaining Connections to Back-End Services [API] : F-OP
40. (Back-End Service Replication [ ]) $\xrightarrow{\text{Help}}$ Availability [API] : NF-OP
41. (API Gateway [ ], Service Registration [ ], Service Discovery [ ], Service Mapping and Composition [ ], Service Orchestration [ ]) $\xrightarrow{\text{or}}$ Server-Side Extensibility [API]: NF-OP
42. (Central Gateway [ ], Multiple Gateways, Back-End for Front-End [ ]) $\xrightarrow{\text{xor}}$ API Gateway [ ]: F-OP
43. (Self-Registration [ ], Third-Party Registration [ ]) $\xrightarrow{\text{xor}}$ Service Registration [ ]: F-OP
44. (Server-Side Service Discovery [ ], Client-Side Service Discovery [ ]) $\xrightarrow{\text{xor}}$ Service Discovery [ ]: F-OP
45. (Server-Side Service Mapping and Composition [ ], Server-Side Service Mapping and Composition [ ]) $\xrightarrow{\text{xor}}$ Service Composition [ ] : F-OP
46. (Server-Side Service Two-Phase Transaction Management [ ], Client-Side Service Two-Phase Transaction Management [ ]) $\xrightarrow{\text{xor}}$ Service Orchestration [ ] : F-OP
47. (Message Format Conversion [ ]) $\xrightarrow{\text{Help}}$ Flexibility in message Format [API]: NF-OP
48. (JSON-XML Convertor [ ]) $\xrightarrow{\text{Help}}$ Message Format Conversion [ ] : F-OP
49. (Interface Translation [ ]) $\xrightarrow{\text{Help}}$ Flexibility in Message Parameters [API]: NF-OP
50. (Protocol Translation [ ]) $\xrightarrow{\text{Help}}$ Flexibility in Communication Protocol [API]: NF-OP
51. (SOAP-REST Translation [ ]) $\xrightarrow{\text{Make}}$ Protocol Translation [ ] : F-OP
52. (One-to-One Communication Style [API], One-to-Many Communication Style [API]) $\xrightarrow{\text{or}}$ Communication Style [API]: F-REF
53. (Synchronous Communication [ ], Asynchronous Communication [ ], Synchronous to Asynchronous Communication [ ]) $\xrightarrow{\text{xor}}$ One-to-One Communication Style [API]: F-OP
54. (Publish and Subscribe [ ]) $\xrightarrow{\text{Make}}$ One-to-Many Communication Style [API]
55. (API-Key [ ]) $\xrightarrow{\text{Make}}$ Access Simplicity [API] : COR

56. (API-Key [ ]) $\xrightarrow{\text{Make}}$ Usage Simplicity [API] : COR
57. (API-Key [ ]) $\xrightarrow{\text{Make}}$ Latency [API] : COR
58. (API-Key [ ]) $\xrightarrow{\text{Help}}$ Access Confidentiality [API] : COR
59. (API-Key [ ]) $\xrightarrow{\text{Break}}$ Message Confidentiality [API] : COR
60. (API-Key [ ]) $\xrightarrow{\text{Break}}$ Privacy [API] : COR
61. (Username and Password [ ]) $\xrightarrow{\text{Some}+}$ Access Simplicity [API] : COR
62. (Username and Password [ ]) $\xrightarrow{\text{Make}}$ Usage Simplicity [API] : COR
63. (Username and Password [ ]) $\xrightarrow{\text{Some}-}$ Latency [API] : COR
64. (Username and Password [ ]) $\xrightarrow{\text{Help}}$ Access Confidentiality [API] : COR
65. (Username and Password [ ]) $\xrightarrow{\text{Some}+}$ Message Confidentiality [API] : COR
66. (Username and Password [ ]) $\xrightarrow{\text{Help}}$ Privacy [API] : COR
67. (Mutual Certificate Based Authentication X.509 [ ]) $\xrightarrow{\text{Break}}$ Access Simplicity [API] : COR
68. (Mutual Certificate Based Authentication X.509 [ ]) $\xrightarrow{\text{Break}}$ Usage Simplicity [API] : COR
69. (Mutual Certificate Based Authentication X.509 [ ]) $\xrightarrow{\text{Some}-}$ Latency [API] : COR
70. (Mutual Certificate Based Authentication X.509 [ ]) $\xrightarrow{\text{Make}}$ Access Confidentiality [API] : COR
71. (Mutual Certificate Based Authentication X.509 [ ]) $\xrightarrow{\text{Make}}$ Message Confidentiality [API] : COR
72. (Mutual Certificate Based Authentication X.509 [ ]) $\xrightarrow{\text{Break}--}$ Privacy [API] : COR
73. (Open Authorization Version 2.0 [ ]) $\xrightarrow{\text{Some}-}$ Access Simplicity [API] : COR
74. (Open Authorization Version 2.0 [ ]) $\xrightarrow{\text{Some}-}$ Usage Simplicity [API] : COR
75. (Open Authorization Version 2.0 [ ]) $\xrightarrow{\text{Some}-}$ Latency [API] : COR
76. (Open Authorization Version 2.0 [ ]) $\xrightarrow{\text{Some}+}$ Access Confidentiality [API] : COR
77. (Open Authorization Version 2.0 [ ]) $\xrightarrow{\text{Some}+}$ Message Confidentiality [API] : COR
78. (Open Authorization Version 2.0 [ ]) $\xrightarrow{\text{Some}+}$ Privacy [API] : COR
79. (OpenID Connect Version 1.0 [ ]) $\xrightarrow{\text{Some}-}$ Access Simplicity [API] : COR
80. (OpenID Connect Version 1.0 [ ]) $\xrightarrow{\text{Some}-}$ Usage Simplicity [API] : COR
81. (OpenID Connect Version 1.0 [ ]) $\xrightarrow{\text{Break}--}$ Latency [API]: COR
82. (OpenID Connect Version 1.0 [ ]) $\xrightarrow{\text{Some}+}$ Access Confidentiality [API] : COR
83. (OpenID Connect Version 1.0 [ ]) $\xrightarrow{\text{Some}+}$ Message Confidentiality [API] : COR
84. (OpenID Connect Version 1.0 [ ]) $\xrightarrow{\text{Some}+}$ Privacy [API] : COR
85. (Spike Arrest [ ]) $\xrightarrow{\text{Some}-}$ Availability [API] : COR
86. (Spike Arrest [ ]) $\xrightarrow{\text{Some}-}$ Latency [API] : COR
87. (Spike Arrest [ ]) $\xrightarrow{\text{Make}}$ Robustness [API] : COR
88. (Spike Arrest [ ]) $\xrightarrow{\text{Some}-}$ Integrity [API] : COR
89. (Consumption Quota [ ]) $\xrightarrow{\text{Some}-}$ Availability [API] : COR

90. (Consumption Quota [ ]) $\xrightarrow{\text{Some+}}$ Robustness [API] : COR
91. (Consumption Quota [ ]) $\xrightarrow{\text{Make}}$ Integrity [API] : COR
92. (Concurrent Rate Limit [ ]) $\xrightarrow{\text{Some−}}$ Availability [API] : COR
93. (Concurrent Rate Limit [ ]) $\xrightarrow{\text{Some−}}$ Latency [API] : COR
94. (Concurrent Rate Limit [ ]) $\xrightarrow{\text{Make}}$ Robustness [API] : COR
95. (Concurrent Rate Limit [ ]) $\xrightarrow{\text{Some−}}$ Integrity [API] : COR
96. (Synchronous Communication [ ]) $\xrightarrow{\text{Break}}$ Throughput [API] : COR
97. (Synchronous Communication [ ]) $\xrightarrow{\text{Make}}$ Latency [API] : COR
98. (Synchronous Communication [ ]) $\xrightarrow{\text{Some−}}$ Robustness [API] : COR
99. (Synchronous Communication [ ]) $\xrightarrow{\text{Some+}}$ Integrity [API] : COR
100. (Asynchronous Communication [ ]) $\xrightarrow{\text{Make}}$ Throughput [API] : COR
101. (Asynchronous Communication [ ]) $\xrightarrow{\text{Some−}}$ Latency [API] : COR
102. (Asynchronous Communication [ ]) $\xrightarrow{\text{Some+}}$ Robustness [API] : COR
103. (Asynchronous Communication [ ]) $\xrightarrow{\text{Some−}}$ Integrity [API] : COR
104. (Synchronous to Asynchronous Communication [ ]) $\xrightarrow{\text{Make}}$ Throughput [API] : COR
105. (Synchronous to Asynchronous Communication [ ]) $\xrightarrow{\text{Some+}}$ Latency [API] : COR
106. (Synchronous to Asynchronous Communication [ ]) $\xrightarrow{\text{Make}}$ Robustness [API] : COR
107. (Synchronous to Asynchronous Communication [ ]) $\xrightarrow{\text{Make}}$ Integrity [API] : COR
108. (Synchronous to Asynchronous Communication [ ]) $\xrightarrow{\text{Some+}}$ Flexibility in Communication Protocol [API] : COR
109. (Publish and Subscribe [ ]) $\xrightarrow{\text{Some+}}$ Access Simplicity [API] : COR
110. (Publish and Subscribe [ ]) $\xrightarrow{\text{Make}}$ Server-Side Extensibility [API] : COR
111. (Publish and Subscribe [ ]) $\xrightarrow{\text{Make}}$ Client-Side Extensibility [API] : COR
112. (Publish and Subscribe [ ]) $\xrightarrow{\text{Help}}$ Access Confidentiality [API] : COR
113. (Publish and Subscribe [ ]) $\xrightarrow{\text{Some−}}$ Response Time [API] : COR
114. (Central Gateway [ ]) $\xrightarrow{\text{Make}}$ Server-Side Extensibility [API] : COR
115. (Central Gateway [ ]) $\xrightarrow{\text{Some−}}$ Throughput [API] : COR
116. (Central Gateway [ ]) $\xrightarrow{\text{Some−}}$ Response Time [API] : COR
117. (Central Gateway [ ]) $\xrightarrow{\text{Some+}}$ Access Confidentiality [API] : COR
118. (Multiple Gateway, Back-End for Front-End [ ]) $\xrightarrow{\text{Make}}$ Server-Side Extensibility [API] : COR
119. (Multiple Gateway, Back-End for Front-End [ ]) $\xrightarrow{\text{Some+}}$ Throughput [API] : COR
120. (Multiple Gateway, Back-End for Front-End [ ]) $\xrightarrow{\text{Some−}}$ Response Time [API] : COR
121. (Multiple Gateway, Back-End for Front-End [ ]) $\xrightarrow{\text{Some+}}$ Access Confidentiality [API] : COR
122. (Third-Party Registration [ ]) $\xrightarrow{\text{Make}}$ Usage Simplicity [API] : COR
123. (Third-Party Registration [ ]) $\xrightarrow{\text{Make}}$ Server-Side Extensibility [API] : COR

124. (Third-Party Registration [ ]) $\xrightarrow{\text{Some+}}$ Access Confidentiality [API] : COR
125. (Third-Party Registration [ ]) $\xrightarrow{\text{Some−}}$ Usage Simplicity [API] : COR
126. (Third-Party Registration [ ]) $\xrightarrow{\text{Some+}}$ Server-Side Extensibility [API] : COR
127. (Third-Party Registration [ ]) $\xrightarrow{\text{Some−}}$ Access Confidentiality [API] : COR
128. (Server-Side Service Discovery [ ]) $\xrightarrow{\text{Make}}$ Usage Simplicity [API] : COR
129. (Server-Side Service Discovery [ ]) $\xrightarrow{\text{Make}}$ Server-Side Extensibility [API] : COR
130. (Server-Side Service Discovery [ ]) $\xrightarrow{\text{Make}}$ Latency [API] : COR
131. (Server-Side Service Discovery [ ]) $\xrightarrow{\text{Some+}}$ Throughput [API] : COR
132. (Server-Side Service Discovery [ ]) $\xrightarrow{\text{Some+}}$ Access Confidentiality [API] : COR
133. (Server-Side Service Discovery [ ]) $\xrightarrow{\text{Some−}}$ Usage Simplicity [API] : COR
134. (Server-Side Service Discovery [ ]) $\xrightarrow{\text{Some+}}$ Server-Side Extensibility [API] : COR
135. (Server-Side Service Discovery [ ]) $\xrightarrow{\text{Some−}}$ Latency [API] : COR
136. (Server-Side Service Discovery [ ]) $\xrightarrow{\text{Some−}}$ Throughput [API] : COR
137. (Server-Side Service Discovery [ ]) $\xrightarrow{\text{Some−}}$ Access Confidentiality [API] : COR
138. (Server-Side API Mapping and Composition [ ]) $\xrightarrow{\text{Make}}$ Usage Simplicity [API] : COR
139. (Server-Side API Mapping and Composition [ ]) $\xrightarrow{\text{Make}}$ Server-Side Extensibility [API] : COR
140. (Server-Side API Mapping and Composition [ ]) $\xrightarrow{\text{Make}}$ Latency [API] : COR
141. (Server-Side API Mapping and Composition [ ]) $\xrightarrow{\text{Some+}}$ Throughput [API] : COR
142. (Server-Side API Mapping and Composition [ ]) $\xrightarrow{\text{Some+}}$ Access Confidentiality [API] : COR
143. (Client-Side API Mapping and Composition [ ]) $\xrightarrow{\text{Some−}}$ Usage Simplicity [API] : COR
144. (Client-Side API Mapping and Composition [ ]) $\xrightarrow{\text{Some+}}$ Server-Side Extensibility [API] : COR
145. (Client-Side API Mapping and Composition [ ]) $\xrightarrow{\text{Some−}}$ Latency [API] : COR
146. (Client-Side API Mapping and Composition [ ]) $\xrightarrow{\text{Some−}}$ Throughput [API] : COR
147. (Client-Side API Mapping and Composition [ ]) $\xrightarrow{\text{Some−}}$ Access Confidentiality [API] : COR
148. (Server-Side Two-Phase Transaction Management [ ]) $\xrightarrow{\text{Make}}$ Usage Simplicity [API] : COR
149. (Server-Side Two-Phase Transaction Management [ ]) $\xrightarrow{\text{Some+}}$ Server-Side Extensibility [API] : COR
150. (Server-Side Two-Phase Transaction Management [ ]) $\xrightarrow{\text{Make}}$ Latency [API] : COR
151. (Server-Side Two-Phase Transaction Management [ ]) $\xrightarrow{\text{Make}}$ Throughput [API] : COR
152. (Server-Side Two-Phase Transaction Management [ ]) $\xrightarrow{\text{Some+}}$ Robustness [API] : COR
153. (Client-Side Two-Phase Transaction Management [ ]) $\xrightarrow{\text{Break}}$ Usage Simplicity [API] : COR
154. (Client-Side Two-Phase Transaction Management [ ]) $\xrightarrow{\text{Some+}}$ Server-Side Extensibility [API] : COR
155. (Client-Side Two-Phase Transaction Management [ ]) $\xrightarrow{\text{Break}}$ Latency [API] : COR
156. (Client-Side Two-Phase Transaction Management [ ]) $\xrightarrow{\text{Break}}$ Throughput [API] : COR

Appendix 4: The API Design Question Bank

	Question Topics	Example Design Questions
1	- “API Evolvability” - “Backward Compatibility with Minor Changes”	“The API needs to be upgraded and newer versions of the API need to be released. The changes are minor and only apply to API some request and response parameters. How to handle compatibility with the current version of the API for the clients who are using the current version of the API.”
2	- “API Evolvability” - “Backward Compatibility with Major Changes”	“The API needs to be upgraded and newer versions of the API need to be released. The changes are major, and the API calls and the API address will also change. How to handle compatibility with the current version of the API.”
3	- “API Security” - “Confidentiality” - “Service Confidentiality” - “Access Control” - “Access Authorization”	- “How to secure access to the API?” - “Design an access authorization mechanism for the API.” - “Design an access control mechanism for the API.”
4	- “API Security” - “Confidentiality” - “Message Confidentiality”	- “How to protect and secure the API from eaves dropping (i.e., unauthorized listening to the API requests and responses.)” - “Design a mechanism that protects the confidentiality of API interactions (requests and responses) with the clients.” - “How to protect and security the API from man in the middle attack (i.e., changing the requests and responses that are communicated between the API and the clients.”
5	- “API Security” - “Privacy” - “Access Privacy”	“How to protect the privacy of the end-user’s data upon access to the API.”
6	- “API Security” - “Privacy” - “Service Privacy”	“How to protect the privacy of the data communicated with or stored by the API?”
7	- “API Security” - “Operational Security” - “Robustness” - “Failure Prevention” - “Congestion Control” - “Throttling”	- “How to secure and protect the API from denial of service attacks (i.e., bombarding the back-end services with huge volumes of API calls so that other clients cannot access and use the API.” - “How to protect the API from failure in the face of bursts in the traffic of API calls during peak times?” - “Design a mechanism to help the API stay robust in the face of bursts in the traffic of API calls during peak times?” - “How to protect the API from failure in providing service during peak times?” - “Design a congestion control / throttling mechanism for the API.”
8	- “API Security” - “Operational Security” - “Robustness” - “Failure Detection”	- “Back-end systems and service may become unavailable momentarily or permanently due to various reasons such as failure, upgrade, or disconnection from the network. Design a mechanism that helps the API to detect failure or unavailability of the back-end services.”
9	- “API Security” - “Operational Security” - “Robustness” - “Failure Prevention”	- “Some Back-end systems and service may become unavailable momentarily or permanently due to various reasons such as failure, upgrade, or disconnection from the network. Design a mechanism that prevents the API from complete failure in responding to the clients’ requests.”
10	- “API Security” - “Operational Security” - “Robustness” - “Failure Recovery”	- “Back-end systems and service may become unavailable momentarily or permanently due to various reasons such as failure, upgrade, or disconnection from the network. Design a mechanism that helps the API manage the situations when the back-end systems cannot respond to the clients.”
11	- “API Security” - “Operational Security” - “Robustness”	- “Malicious clients may attack the API and the back-end services and corrupt the healthy operation of the API. Design a mechanism that helps the API be robust against potential attacks or malicious usage.”

	Question Topics	Example Design Questions
	- “Threat Detection”	
12	- “API Security” - “Operational Security” - “Traceability”	“Malicious clients may steal the identity of the original clients and find access to the API. Design a mechanism that allows the API to trace identify theft attacks, and to trace the interactions with malicious clients.”
13	- “API Performance” - “Throughput” - “Load balancing” - “Load distribution”	- “Design a mechanism that allows the API to gracefully manage and handle high volumes / load of API calls per second.” - “How to handle high loads of API calls per second?” - “Design a load balancing (or load distribution) mechanism for the API?”
14	- “API Performance” - “Response Time” - “Caching”	- “The API should respond to (numerous) clients in real-time (or timely manner). Design a mechanism that helps minimize the response of the API?” - “Design a caching mechanism for the API.”
15	- “API Performance” - “Response Time” - “Traffic Prioritization”	- “The API should respond to the clients in real-time. Design a mechanism that allows the API to control its response time.”
16	- “API Performance” - “Availability”	- “The API should be highly available. Design a mechanism that allows the API to be available even in the presence of failure in the back-end servers (services).”
17	- “API Extensibility” - “Server-Side Extensibility”	- “Clients may request to upgrade or down-grade the provided service. Design a mechanism that allows easily to scale up or scale down the service of the API; i.e., to add or remove back-end services easily.”
18	- “API Extensibility” - “Server-Side Extensibility”	- “Clients may request to upgrade or down-grade the provided service. Design a mechanism that allows to easily scale up or scale down the service of the API; i.e., to add or remove back-end services easily.”
19	- “API Extensibility” - “Server-Side Extensibility” - “Service Registry”	- “New service providers (or servers, data providers, or data sources) can register themselves as a back-end service for the API. Design a service registry mechanism for the API.” - “Design a mechanism that allows to dynamically *add back-end services* or servers to the API over time.”
20	- “API Extensibility” - “Server-Side Extensibility” - “Service Discovery”	- “The back-end service providers (data sources) (or their location) change over time. Moreover, some back-end services may be unavailable in a specific time or be disconnected from the network. Design a mechanism that allows the API provider to find related back-end services.” - “How to handle dynamic addition and removal of the back-end services.”
21	-“API Extensibility” - “Server-Side Extensibility” - “Service Composition”	“The API uses a combination of other APIs to respond to the clients’ requests. How should the API be designed?”
22	- “API Extensibility” - “Server-Side Extensibility” - “Service Orchestration”	“The API cooperates with other APIs to respond to the clients’ requests. How the API should be designed?”
23	- “API Communication Style” - “Interaction Mechanism”	- “Design a communication style between the API and its clients.” - “How should the API notify the clients of the updates?”
24	-“API Interoperability” -“Flexibility in data format”	“The API should be able to work with different types of clients and with varying data formats (e.g. (JSON, XML, or HTML)). How to handle flexibility in data format?”
25	-“API Interoperability” -“Flexibility in communication protocol”	“The API should work with different types of clients with various communication protocols (e.g. RESP, or SOAP). How to handle flexibility in communication protocol?”
26	-“API Interoperability” -“Flexibility in message parameters”	“The API should work with clients having varying message formats (i.e, message header and query parameters). How to handle flexibility in message parameters?”

Appendix 5: The Web API Design Exam and the Answers Provided by RAPID

1. A Banking Platform – The Payment API

API Specification

The Payment API enables online payment and allows to transfer fund from a given customer's account to a destination account. The transfer is initiated by a client application or service. Access to the destination account is provided by a third-party account API provider.

The Payment API will be used in different scenarios such as online purchases (e.g. a merchant site that integrates secure bank payment).

Functional Requirements

- The client provides source account, destination account number, and the transfer amount. The API performs transfer an amount of fund from the source account to the destination account.

Non-Functional Requirements

- *Confidentiality and Privacy are Very Critical*: Since the Payment API performs critical transactions on customers' accounts, confidentiality and privacy are very critical requirements to the Payment API. The API should be immune against unauthorized access, and security attacks, such as (identity theft, man in the middle attack, and eaves dropping attacks).
- *Operational Security and Robustness are Very Critical*: Correct and successful operation of the Payment API is very critical. The payment API should work correctly even in the case of failure of back-end services (for example, the third-party account provider API (the destination account provider) may be unavailable for some time, but the payment transaction should work correctly.
- *Availability is Very Critical:* Since the online payment service is used in daily purchases and payment scenarios, it should be highly available (It should provide service 24 * 7 and have 99.95% uptime).
- *Interoperability is Critical:* The payment API should be API to work with various clients having different communication protocols (such as SOAP or REST).

System / Platform	**Banking Platform**
Name	**Payment API**
Type of API	**Public (Open) Web API**
Functionality / Data Provided by the API	Performing online fund transfer between two accounts.
Important Requirements and Priority	(1) Confidentiality of the API is very critical. (2) Privacy of the API is very critical. (3) Robustness of the API is very critical. (4) Availability of the API is critical. (5) Interoperability of the API is critical.

1 **Question.** Design an access authorization mechanism for the Payment API.

Question Topic. Access Authorization [API]

RAPID's Answer. **Either (a)** Username and Password, **or (b)** Open Authorization Version 2.0, **or (c)** OpenID Connect Version 1.0.

Rating: Acceptable-Identical: 2, **Acceptable-Alternative:** 1, **Partly-Unacceptable:**3**, Don't Know:** 1

Status: Unacceptable *

2 **Question.** Design a mechanism that protects the confidentiality of the Payment API interactions (requests and responses) with the clients (the mechanism should protect the Payment API from eaves dropping and unauthorized listening to the API interactions with the clients.)

Question Topic. Message Confidentiality [API]

RAPID's Answer. (a) Secure communication channels **or (b)** Message Encryption.

Rating: Acceptable-Identical: 3, **Acceptable-Alternative:** 1, **Partly-Unacceptable:** 3

Status: Acceptable

3 **Question.** "Design a communication style between the Payment API and its clients."

Question Topic. Communication Style [API]

RAPID's Answer. Synchronous to Asynchronous Communication Style

Rating: Acceptable-Identical: 2, **Acceptable-Alternative:** 1, **Partly-Unacceptable:** 2, **Unacceptable**: 1, **Insufficient information**: 2

Status: Unacceptable *

4 **Question**. "Malicious clients may steal the identity of the original clients and find access to the Payment API. Design a mechanism that allows the payment API to trace identify theft attacks."

Question Topic. Traceability [API]

RAPID's Answer. (a) Activity Logging **or (b)** User Auditing.

Rating: Acceptable-Identical: 1, **Acceptable-Alternative:** 2, **Partly-Unacceptable:** 2, **Unacceptable**: 1, **Insufficient information**: 1

Status: Unacceptable *

5 **Question.** The Payment API should be able to work with different types of clients and with varying data formats (e.g. (JSON, XML)). How to handle flexibility in message format?

Question Topic. Flexibility in Message Format [API]

RAPID's Answer. Message format conversion, JSON-XML Convertor

Rating: Acceptable-Identical: 2, **Acceptable-Alternative:** 2, **Partly-Unacceptable:** 1, **Unacceptable**: 2

Status: Acceptable

6 **Question.** The Payment API cooperates with a third-party account API to transfer money to the destination account. How to design the cooperation between the Payment API and the third-party account API to perform a transfer transaction?

Question Topic. Service Orchestration [ ]

RAPID's Answer. Server-Side Two-Phase Transaction Management.

Rating: Acceptable-Identical: 3, **Acceptable-Alternative:** 1, **Partly-Unacceptable:** 1, **Don't Know**: 2

Status: Acceptable

7 **Question.** The third-party account API (i.e., the destination account provider) may become unavailable momentarily due to various reasons such as failure or disconnection from the web. Design a mechanism that helps the payment API to detect failure or unavailability of the destination account provider.

Question Topic. Failure Detection [API]

RAPID's Answer. **Either (a)** Circuit Breaker (Pattern) **or (b)** Response Time-Outs.

Rating: Acceptable-Identical: 3, **Acceptable-Alternative:** 3, **Partly-Unacceptable:** 1

Status: Acceptable

2. A Weather Map Platform – The Current Weather API

API Specification

The Current Weather API provides data about the current weather of more than 200,000 cities around the world. Weather data of the cities are frequently updated based on global models and data coming from more than 40,000 weather stations which are dispersed geographically. Weather stations and data sources providers can openly register themselves as a weather data provider and they can be added or removed dynamically over time.

The Weather API has different kinds of clients, including unknown applications and services which can access the API for free as well as customers and partner applications who pay for the use of API. Different clients have different limits for calling the API per day.

Functional Requirements

- Given the city name or the geographical coordinates of the city or the zip code of the city, the API provides data about the current weather of a given city.

Non-Functional Requirements

- *Fast Response Time and Low Latency are Very Critical*: The Current Weather API should respond to the clients' requests fast and immediately in the order of milliseconds).
- *Extensibility of the Back-End Data Sources is Critical*: The back-end data sources (weather data providers) can be added and removed dynamically over time. New weather data providers can register themselves over time and provide weather measurement data. The weather data about a given city can be provided by several weather stations.
- *High Throughput is Critical*: The Current Weather API should be able to manage and responds to high volumes of the API calls per second. (Assume that the Weather API should handle more than 30000 API calls per second.)
- *Usage Simplicity is Important:* The API should be simple and utterly convenient to use.
- *Flexibility in Message Parameters is Important:* The current weather API should be flexible towards the message and data formats of different clients as well as their request parameters. For example, the clients may ask for the current weather of a city by providing the city name as the request parameter, or ask the weather by proving its longitude and latitude, or by zip code.

System / Platform	Weather Map Platform
API Name	The Current Weather API
Type of API	Public (Open) Web API
Functionality / Data Provided by the API	Providing Weather Data of a given city
Important Requirements	(1) Response Time of the API is very critical. (2) Latency of the API is very critical. (3) Throughput of the API is critical. (4) Server-Side Extensibility of the API is critical. (5) Usage Simplicity of the API is important. (6) Flexibility in message parameters of the API is important.

1 **Question.** Design an access control mechanism for the Current Weather API. Consider that the Current Weather API has different types of clients with different limits for API calls per day.

Question Topic. Access Control [API]

RAPID's Answer. API-Key

Rating: Acceptable-Identical: 4, **Acceptable-Alternative:** 1, **Partly-Unacceptable:** 2

Status: Acceptable

2 **Question**. Design a mechanism that allows the Current Weather API to gracefully handle 30000 API calls per second.

Question Topic. Throughput [API]

RAPID's Answer. (a) Back-end service concurrency **and (b)** Load distribution and balancing.

For load distribution: Either **(a)** Round-Rubin Distribution, **or (b)** Weighted Round Rubin Distribution, **or (c)** Least Connection Distribution, **or (d)** Weighted Least Connection Distribution, **or (e)** Random Load Distribution.

Rating: Acceptable-Identical: 2, **Acceptable-Alternative:** 2, **Partly-Unacceptable:** 3

Status: Acceptable

3 **Question.** How to protect the Current Weather API from denial of service failures that might occur due to congestion in the traffic of clients' requests?

Question Topic. Congestion Control [API]

RAPID's Answer. Consumption Quota or Rate Limit.

Rating: Acceptable-Identical: 4, **Acceptable-Alternative:** 1, **Partly-Unacceptable:**1, **Unacceptable:**1

Status: Acceptable

4 **Question.** The Current Weather API should respond to numerous clients in real-time. How to minimize the response time of the API?

Question Topic. Response Time [API]

RAPID's Answer. (a) Caching responses of API **or (b)** Caching and Maintaining Connections to Back-end services.

For caching responses: Either **(a)** Caching most frequent responses**, or (b)** Caching most recent responses, **or (c)** Caching most Probable Reponses.

For caching and maintaining connections to the back-end services: Connection Pooling**.**

Rating: Acceptable-Identical: 4, **Acceptable-Alternative:** 2, **Partly-Unacceptable:**1

Status: Acceptable

5 **Question**. Design a mechanism that allows to easily *add or remove back-end weather data providers* (weather stations) for the Weather API over time.

Question Topic. Server-Side Extensibility [API]

RAPID's Answer. Third-party Service Registration.

Rating: Acceptable-Identical: 5, **Acceptable-Alternative:** 1, **Partly-Unacceptable:**1

Status: Acceptable

6 **Question**. The network location (IP address) of weather data providers changes over time. Moreover, there may be several weather data providers that can provide the data about the weather of a given city. How to find related back-end weather data providers?

Question Topic. Service Discovery and Routing [ ]

RAPID's Answer. Server-side service discovery and routing.

Rating: Acceptable-Identical: 3, **Acceptable-Alternative:** 1, **Partly-Unacceptable:** 3

Status: Acceptable

7 **Question.** The Current Weather API should be able to work with different types of clients and with various message parameters (i.e., varying query parameters (such as city name, zip code, or geographical coordinates). How to handle flexibility in message parameters?

Question Topic. Flexibility in message parameters [API]

RAPID's Answer. Adapter.

Rating: Acceptable-Identical: 4, **Acceptable-Alternative:** 3

Status: Acceptable

3. A Cloud Storage Platform – The Write Bucket / Object API

API Specification

The Write Bucket / Object API allows to write a bucket or an object on a cloud storage service.

Functional Requirements

- Given the ID of a bucket or an object and its content, the API writes or updates the bucket or object on the storage system.

Non-Functional Requirements

- *High Throughput is Very Critical:* The Write API should be able to handle high-loads of API calls (Assume that the storage service should be able to handle 100 clients simultaneously, each of which can make 1000 API per second. Hence, the API should be able to handle 100*1000= 100,000 requests per second).
- *Privacy of Clients' Data is Very Critical*: Since the stored data belong to the clients, the privacy of the stored objects should be protected on the back-end data storage servers. The clients' data should not be visible to unintended and unauthorized audience.
- *High Availability is Very Critical:* The Write API should meet 99.5 % uptime.
- *Operational Security and Robustness is Very Critical*: The write and update operation should work correctly even in the face of failure of the back-end systems (e.g. data storage servers may fail during a write request). The write API should also be immune against corruption in the back-end data storage systems (the storage system may fail permanently). The written objects and buckets should not be corrupted or missed over time.
- *Server-Side Extensibility is Very Critical*: The clients may need to scale up or down (i.e., upgrade, or down grade) their space for writing and storing objects. The API should allow the client to seamlessly scale up or down the required storage service.
- *Access Confidentiality is Critical*: The Write API should only be accessed by the client whose files and content are stored on the storage system.

System / Platform	A Cloud Storage Platform
API Name	The Write Bucket / Object API
Type of API	Protected (Partner to Partner) Cloud API
Functionality / Data Provided by the API	Returning back a bucket of stored objects
Important Requirements	(1) Throughput of the API is Very Critical. (2) Privacy of the API is Very Critical. (3) Availability of the API is Very Critical. (4) Robustness of the API is Very Critical. (5) Server-Side Extensibility of the API is Very Critical. (6) Access Confidentiality of the API is Critical.

1 **Question:** How to protect the privacy of the clients' data that are received and stored by the Write API? The stored data on the cloud storage belongs to the clients. The storages servers should preserve the privacy of the users' data.

Question Topic: Privacy [API]

RAPID's Answer: (a) End-User Notification and Approval **or (b)** Data Masking.

Rating: Acceptable-Alternative: 2, **Partly-Unacceptable:** 2, **Unacceptable**: 2, **Insufficient Information:** 1

Status: Unacceptable *

2 **Question:** Design a mechanism that allows the Write API to gracefully manage and handle 100,000API calls per second.

Question Topic: Throughput [API]

RAPID's Answer: (a) Back-end service concurrency**, and (b)** Load distribution and balancing.

For load distribution use: **Either (a)** Round-Rubin Distribution, **or (b)** Weighted Round Rubin Distribution, **or (c)** Least Connection Distribution, **or (d)** Weighted Least Connection Distribution, **or (e)** Random Load Distribution.

Rating: Acceptable-Identical: 3**, Acceptable-Alternative:** 1, **Partly-Unacceptable:** 2, **Don't Know:** 1

Status: Acceptable

3 **Question:** The Write API should be highly available. Design a mechanism that allows the API to be available even in the presence of failure in the back-end data storage servers.

Question Topic: Availability [API]

RAPID's Answer: Back-End Service Replication.

Rating: Acceptable-Identical: 4, Partly-Unacceptable: 1, **Insufficient Information:** 1, **Don't Know:** 1

Status: Acceptable

4 **Question:** Clients may request to upgrade or down-grade the storage service according to their storage needs. Design some mechanism that allows easily to scale up or scale down the service of the API; i.e., to add or remove back-end services easily.

Question Topic: Server-Side Extensibility [API]

RAPID's Answer: (a) API Gateway **or (b)** Service Registration **or (c)** Service Discovery **or (d)** Service Mapping and Composition**, or (e)** Service Orchestration.

Rating: Acceptable-Identical: 2, **Acceptable-Alternative:** 1, **Partly-Unacceptable:** 2, **Unacceptable:** 2

Status: Unacceptable*

5 **Question:** Back-end data sources may become unavailable momentarily or permanently due to various reasons such as failure. Design a mechanism that helps the Write API to proactively detect failure or unavailability of the back-end systems before complete failure.

Question Topic: Failure Detection [API]

RAPID's Answer: Either (a) Circuit Breaker (Pattern) **or (b)** Response Time-Outs.

Rating: Acceptable-Identical: 3, **Acceptable-Alternative:** 1, **Partly-Unacceptable:** 2, **Unacceptable:** 1

Status: Acceptable

6 **Question:** Malicious clients may attack the write API and the back-end services and corrupt the healthy operation of the API. Design a mechanism that helps the write API be robust against potential attacks or malicious usage.

Question Topic: Robustness [API]

RAPID's Answer: Traffic Monitoring. In Traffic monitoring **(a)** detect unusual request loads, **or (b)** detect unusual request patterns.

Rating: Acceptable-Identical: 2, **Acceptable-Alternative:** 2, **Partly-Unacceptable:** 1, **Unacceptable:** 1, **Don't Know:** 1

Status: Acceptable

4. A Vehicle Control Platform – The Lock Status API

API Specification

The Lock Status API provides data about the lock status of a vehicle; i.e., It informs the clients whether the vehicle doors are locked or not. Remote client applications can connect to the Lock API via the vehicle WiFi and WLAN to check the lock status.

Functional Requirements

- Given the Vehicle ID, the API returns the lock status of the related vehicle.

Non-Functional Requirements

- *Access and Message Confidentiality is Very Critical*: The Lock Status API should only be accessed by the clients who are authorized by the driver or owner of the vehicle. Moreover, the requests and responses of the lock status API should be immune against eavesdropping (i.e., unauthorized listening to the API interactions).
- *Low Latency is Critical*: The Lock Status API should respond to the clients in real-time and in a timely manner.
- *Flexibility in Message and Data format is Important*: The Lock Status API should be able to provide lock status data in different formats, such as JSON, XML, HTML and should be able to work with client applications having various message formats.
- *Evolvability is Important*: The Lock Status API may evolve over time and newer versions of the API with different message (request and response) parameters will be available over time.

System / Platform	A Vehicle Control Platform
API Name	The Lock Status API
Type of API	Open Web API
Functionality / Data Provided by the API	Informing the clients of the lock status of a vehicle.
Important Requirements	(1) Access Confidentiality of the API is Very Critical. (2) Message Confidentiality of the API is Very Critical. (3) Latency of the API is Critical. (4) Flexibility in message and data format of the API is important. (5) Evolvability of the API is important.

1 **Question:** Design an access authorization mechanism for the Lock Status API.
Question Topic: Access Authorization [API]
RAPID's Answer: Mutual Certificate-Based Authentication X .509.
Rating: Acceptable-Identical: 5**, Partly-Unacceptable:** 2
Status: Acceptable

2 **Question.** Design a mechanism that protects the confidentiality of Lock Status API interactions (requests and responses) with the clients.
Question Topic: Message Confidentiality [API]
RAPID's Answer: (a) Secure communication channels **or (b)** Message Encryption (request and response encryption).
Rating: Acceptable-Identical: 5**, Acceptable-Alternative:** 2
Status: Acceptable

3 **Question.** Newer versions of the Lock Status API will be released over time that might differ in the message parameters. How to handle compatibility with the current version of the API.
Question Topic: Compatibility with Minor Changes [API]
RAPID's Answer. Adapter.
Rating: Acceptable-Identical: 1**, Acceptable-Alternative:** 5, **Partly-Unacceptable:** 1
Status: Acceptable

5. A Flight Data Platform –The Flight Time Table API

API Specification

The Flight Time Table API provides data about the status and schedule of any flight around the world. The flight information includes take-off information and all the involved airport arrivals and departure times. The data comes from the various airlines and airports. The Flight Time Table API is used by various clients, such as travel booking agencies.

Functional Requirements

- Given a flight number, the API returns the status and timetable of the flight.

Non-Functional Requirements

- *High Availability is Critical*: The API is used in many business scenarios and should be available 99.95% of the times.
- *Real-Time Response Time is Critical*: The API should respond to the clients immediately in the order of milliseconds.
- *Robustness is Critical:* The API should be immune against the failure of the back-end data providers. Some back-end data source providers may be disconnected in a specific query time.
- *High Throughput is Important*: The Flight Time Table API should be able to handle up to 15000 requests per second.

System / Platform	A Flight Data Platform
API Name	The Flight Time Table API
Type of API	Public (Open) Web API
Functionality / Data Provided by the API	Informing the clients of the schedule of a given flight
Important Requirements	(1) Availability of the API is critical. (2) Response time of the API is critical. (3) Robustness of the API is critical. (4) Throughput of the API is important.

1 **Question:** The Flight Time Table API should respond to numerous clients in real-time (or timely manner). Design a mechanism that helps minimize the response time of the Flight Time Table API?
Question Topic: Response Time [API]
RAPID's Answer: (a) Caching responses of API **or (b)** Caching and Maintaining Connections to Back-end services.
For caching responses use: Either **(a)** Caching most frequent responses **or (b)** Caching most recent responses, **or (c)** Caching most probable Reponses.
For caching and maintaining connections to the back-end services use: Connection pooling**.**
Rating: Acceptable-Identical: 5**, Acceptable-Alternative:** 2
Status: Acceptable

2 **Question:** Back-end data sources of the Flight Time Table [API] may anyway become unavailable momentarily. Design a mechanism that helps the Flight Time Table API manage the situations when the back-end systems cannot respond to the clients.
Question Topic: Failure Recovery [API]
RAPID's Answer: Provide fall backs: Either **(a)** Return Empty Responses **or (b)** Return Cached Responses.
Rating: Acceptable-Alternative: 2**, Partly-Unacceptable:** 4, **Unacceptable:** 1
Status: Unacceptable *

3 **Question.** How to protect the Flight Time Table API from denial of service failures that might occur due to congestion in the traffic of clients' requests?
Question Topic: Congestion Control [API]
RAPID's Answer: (a) Concurrent Rate Limit **or (b)** Spike Arrest.
Rating: Acceptable-Identical: 3**, Acceptable-Alternative:** 3**, Partly-Unacceptable:** 1
Status: Acceptable

4 **Question:** How to meet 99.95% uptime?
Question Topic: Availability [API]
RAPID's Answer: Back-end Service Replication.
Rating: Acceptable-Identical: 2**, Acceptable-Alternative:** 2, **Partly-Unacceptable:** 1, **Insufficient Information:** 1, **Don't Know:** 1
Status: Acceptable

6. A Social Network Platform – The Friends API

API Specification

The Friends API provides the list of friends of a given user to the third-party clients. The API is protected and provides service to partner businesses and enterprise.

Three possible use cases of the Friend API are as follows: (a) to advertise or recommend the products that a user buy to the friends of the user; (b) to show what friends are watching or buying on a web-site; (c) to use a third-party service to search within the list of the friends of a user.

Functional Requirements

- Given a users' name or ID, the API provides the list of the friends of the user.
- The client may require access the friends list of a user more than one. The client may want to be notified of new friends, when a friend is added.

Non-Functional Requirements

- *Access Confidentiality is Very Critical*: The Friends API should only be accessible to authorized clients.
- *Privacy is Very Critical*: Friends data is private and belongs to the user. The confidentiality of the friend data is highly important and must be preserved. The Friends API should not allow access to a user's data without the consent of data owner. The API must not reveal the user's data to unwanted audience.
- *Short Latency is Critical*: The API should respond to the clients' request in a timely manner.

System / Platform	A Social Network Platform
Name	The Friends API
Type of API	Protected (Partner to Partner) Web API
Functionality / Data Provided by the API	- Providing users' friends List -Providing more than once access to the clients
Important Requirements	(1) Privacy of the API is very critical. (2) Access Confidentiality of the API is very critical. (3) Latency of the API is critical.

1 **Question:** How to protect the privacy of the end-user's data upon clients' access to the Friends API. Consider the case that the client may need more than once access to the Friends API.

Question Topic: Privacy [API]

RAPID's Answer: (a) End-User's Notification and Approval Upon API Access **or (b)** Data Masking.

Rating: Acceptable-Identical: 1**, Acceptable-Alternative:** 2**, Partly-Unacceptable:** 1**, Unacceptable:** 2, **Don't Know**: 1

Status: Unacceptable *

2 **Question:** Design an access authorization mechanism for the Friends API. Consider the case that the client may need more than once access to the Friends API.

Question Topic: Access Authorization [API]

RAPID's Answer: Either (a) Username and Password **or (b)** Open Authorization Version 2.0.

Rating: Acceptable-Identical: 2**, Partly-Unacceptable:** 2**, Insufficient Information:** 1, **Don't Know**: 2

Status: Unacceptable *

7. A Social Network Platform – The Filter and Track Posts API

API Specification

The Track Posts API notifies the clients of the users' posts in which they are interested.

One use case of the Track Posts API is to fetch users' posts with certain keywords from the social network platform and show the messages on a third-party application or web-site. For example, posts which contain the name of a product.

Functional Requirements

- The API sends the posts of certain topics to the clients who are interested in that topic.

Non-Functional Requirements

- *Low Latency is important*: Real-Time notification of the clients of a post of interest is important.
- *Reliability is important*: The Filter and Track Posts API should guarantees notifying the client of new updates, meaning that if any new status is posted in which a client is interested, they client will be notified of the new post.

System / Platform	A Social Network Platform
API Name	The Filter and Track Posts API
Type of API	Public (Open) Web API
Functionality / Data Provided by the API	Informing the clients of posts in which they are interested.
Important Requirements	(1) Latency of the API is important. (2) Reliability of the API is important.

1 **Question:** How should the Filter and Track Post API notify the clients of the posts in which they are interested?

Question Topic: Communication Style [API]

RAPID's Answer: Publish and Subscribe

Rating: **Acceptable-Identical**: 5, **Partly-Unacceptable**: 1, **Unacceptable**: 1

Status: Acceptable