

Interface Evolution Patterns – Balancing Compatibility and Extensibility across Service Life Cycles

Daniel Lübke
iQuest GmbH, Hanover, Germany

Olaf Zimmermann
University of Applied Sciences of
Eastern Switzerland, Rapperswil,
Switzerland

Cesare Pautasso
Software Institute, Faculty of
Informatics, USI Lugano, Switzerland

Uwe Zdun
University of Vienna, Faculty of
Computer Science, Software
Architecture Research Group, Austria

Mirko Stocker
University of Applied Sciences of
Eastern Switzerland, Rapperswil,
Switzerland

ABSTRACT

Remote Application Programming Interfaces (APIs) are technology enablers for distributed systems and cloud-native application development. API providers find it hard to design their remote APIs so that they can be evolved easily; refactoring and extending an API while preserving backward compatibility is particularly challenging. If APIs are evolved poorly, clients are critically impacted; high costs to adapt and compensate for downtimes may result. For instance, if an API provider publishes a new incompatible API version, existing clients might break and not function properly until they are upgraded to support the new version. Hence, applying adequate strategies for evolving service APIs is one of the core problems in API governance, which in turn is a prerequisite for successfully integrating service providers with their clients in the long run. Although many patterns and pattern languages are concerned with API, service design, and related integration technologies, patterns guiding the evolution of APIs are missing to date. Extending our emerging pattern language on Microservice API Patterns (MAP), we introduce a set of patterns focusing on API evolution strategies in this paper: API Description, Version Identifier, Semantic Versioning, Eternal Lifetime Guarantee, Limited Lifetime Guarantee, Two in Production, Aggressive Obsolescence, and Experimental Preview. The patterns were mined from public Web APIs and industry projects the authors had been involved in.

CCS CONCEPTS

• **Software and its engineering** → **Patterns**; *Designing software*;

ACM Reference Format:

Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. 2019. Interface Evolution Patterns – Balancing Compatibility and Extensibility across Service Life Cycles. In *24th European Conference on Pattern Languages of Programs (EuroPLoP '19)*, July 3–7, 2019, Irsee, Germany. ACM, New York, NY, USA, 24 pages. <https://doi.org/10.1145/3361149.3361164>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroPLoP '19, July 3–7, 2019, Irsee, Germany

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6206-1/19/07...\$15.00

<https://doi.org/10.1145/3361149.3361164>

1 INTRODUCTION

Application Programming Interfaces (APIs) are found in every programmable platform for developing and running many different applications. Such APIs allow end user applications to configure, access, and reuse the underlying platform features and services in a developer-friendly way, which eases the construction of such applications. Platforms thrive when developers flock to them and write applications for them, and thus bring valuable users and their traffic [27]. Distributed system architectures, such as service-oriented architectures [15], cloud-native applications [10], and microservices [30], offer *remote APIs* for their software components (called services). Today, more and more software services are delivered over the Web [2]. As a consequence, remote API design for services becomes more and more crucial for most (if not all) existing and new software-intensive systems.

To be successful, APIs should expose well-defined, stable contracts that provide a baseline for building applications on top of them (in terms of expectations and delivery guarantees). Designing API well has been the focus of two previously published categories of our MAP pattern language [23, 31]¹.

However, APIs must also be maintained and – like all software – it has to *evolve*, for instance to be able to fix bugs and add features. Such evolution of an API requires that API providers and clients – who usually follow different life cycles [18] – establish rules and policies to ensure that a) the provider can improve and extend the API and its implementation, and b) the client can keep functioning with no or few required changes as long as possible. Modifying an API might lead to client-breaking changes. However, breaking changes should be minimized as they cause migration efforts for a potentially large and sometimes unknown number of clients. If such changes are required and cause API version upgrades, they have to be communicated and handled well in order to reduce the associated risk and cost.

The topic of API evolution has been brought up frequently in workshops that we had with industry partners. Their input as well as the collection and analysis of more than 30 Web APIs and API-related specifications led to the API evolution patterns presented in this paper. Our previous work provides information about pattern sources and mining approach [28, 31].

¹All patterns from the MAP language published so far can be found on the website <https://microservice-api-patterns.org/>.

API providers and clients have to balance different, conflicting concerns in order to follow their own life cycles; a certain amount of autonomy is required to avoid a tight coupling between them. In response, the patterns presented in this paper jointly support API owners, designers, and users when answering the following question:

What are the governing rules balancing stability and compatibility with maintainability and extensibility during API evolution?

During our analysis, we mined the following patterns: An *API Description* can be used to initially specify the API and provide a mechanism to not only define syntactical structure (a.k.a. the technical contract), but also cover organizational matters such as ownership, support, and evolution strategies. The *Version Identifier* pattern defines how version numbers can be explicitly transmitted in APIs in order to indicate the API version used. It can be used in conjunction with the *Semantic Versioning* pattern, which describes a way in which compound version numbers can be defined in order to express compatibility and impact of functional changes. If new API versions are developed and deployed in production, there are different strategies for introducing the new version and decommissioning old ones: *Two in Production* defines how many incompatible versions should be kept active at the same time. *Limited Lifetime Guarantee* establishes a fixed time range for the lifetime of an API version after its initial deployment. Finally, when following the *Eternal Lifetime Guarantee* pattern, the API provider has the burden to keep backwards compatibility forever. The *Aggressive Obsolescence* pattern can be used for phasing out existing features as early as possible. To ease the design of new APIs, gain experience, and gather feedback, the *Experimental Preview* pattern can be applied to indicate that no API availability and support guarantees are given, but the API can be used opportunistically.

The paper is structured in the following way. Section 2 discusses relations to patterns in other languages. Section 3 outlines the basic abstractions that are used throughout the paper and the overall language organization. Section 4 then describes the API Evolution Patterns in detail. Section 5 concludes and gives a brief outlook on future work.

2 RELATIONS TO OTHER PATTERNS AND PATTERN LANGUAGES

We discussed pattern languages that deal with distributed system and API design in our previous two EuroPLoP papers in detail [23, 31], including Remoting Patterns [26], Enterprise Integration Patterns (EIP) [13], Patterns of Enterprise Application Architecture such as Service Layer, Remote Facade, Data Transfer Object (DTO) [11], POSA vol. 4 distributed systems patterns [5], service design patterns by Daigenau [6], and microservices pattern [22]. While some of these sources introduce a similar notion of API contract as proposed here in the *API Description* pattern (see e.g. [5, 26]), the other evolution concerns mentioned above are not (or only implicitly) covered in most of these pattern languages. Managed Evolution [18] shares general information on service governance and versioning.

Our own prior work on our pattern language dealt with APIs explicitly, but have not had a specific focus on evolution aspects either. The patterns introduced in this paper are an extension of

these prior works: In particular, we presented Interface Representation Patterns [31] that introduce five basic patterns for structuring messages in remote APIs: *Atomic Parameter*, *Atomic Parameter List*, *Parameter Tree*, *Parameter Forest*, and *Pagination*. In addition, we presented patterns that deal with runtime qualities and with the communication of API qualities (between provider and client) [23]. Among others, we described how to write *Service Level Agreements* that articulate API qualities in a specific, measurable, and agreed upon way and how to define a *Wish List* allowing the client to select the data to be transmitted in a message and thus gain control over resource usage.

The Service Design Patterns book [6] has an evolution category. Four patterns in this category, *Breaking Changes*, *Versioning*, *Tolerant Reader*, *Consumer-Driven Contracts*, deal with evolution of service contracts and their implementation explicitly. The remaining two patterns *Single Message Argument* and *Dataset Amendment* deal with message construction and structure (in our terminology). As the message structure is defined and exposed in an *API Description*, instances of these two patterns have to be evolved and are therefore subject to versioning. The book was published before microservices became popular as an implementation approach for service-oriented architectures. All service design patterns from the book remain relevant when designing microservice-based APIs. They complement the ones described in this paper.

HTTP resource APIs are one of the particularly popular implementation technologies for microservice APIs. Chapter 13 of the RESTful Web Services Cookbook [24] is devoted to versioning in the context of RESTful HTTP; it presents seven related recipes: "How to Maintain URI Compatibility", "How to Maintain Compatibility of XML and JSON Representations", "How to Extend Atom", "How to Maintain Compatibility of Links", "How to Implement Clients to Support Extensibility", "When to Version" and "How to Version RESTful Web Services". The latter two recipes can be seen as technology-specific realizations of the *Version Identifier* and *Semantic Versioning* patterns; the other ones provide complementary advice.

3 BASIC ABSTRACTIONS AND LANGUAGE ORGANIZATION

This paper uses a number of basic abstractions which form the domain model of our pattern language. At the most abstract level, there are two kinds of *communication participants* (or participants for short) that communicate via an *API*: the *API provider* and the *API client*. An *API provider* exposes any number of *APIs*; an *API client* uses any number of *APIs*. One participant can also play both roles (for instance, in an *API Gateway* [21] in which the *communication participant* offers services as the provider of the gateway and is client to the services shielded by the gateway).

In the client role, a *communication participant* uses *API endpoints* to access the *API*. An *API endpoint* is a provider-side end of a communication channel that specifies the location of the *API* service resources so that *APIs* can be accessed by *API clients*. Each endpoint thus needs to have a unique *address* such as a Uniform Resource Locator (URL), as commonly used on the World-Wide Web, as well as in HTTP-based SOAP or RESTful HTTP. Each *API endpoint* belongs to one *API*; one *API* can have different endpoints.

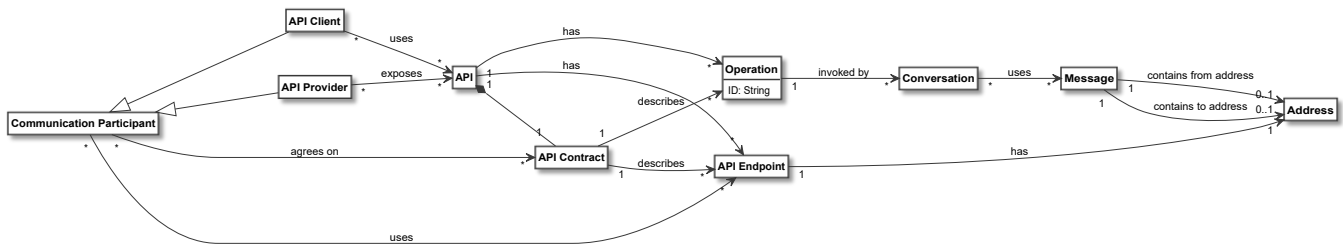


Figure 1: API Domain Abstractions and their Relations

The *API* exposes *operations*. In addition to the endpoint address, an operation identifier is needed to identify the operation. For instance, in SOAP this is the top-level XML tag in the body of the message (if WSDL is used to describe the service); in RESTful HTTP this is the name of the HTTP method (or verb) such as GET or POST. The *operations* of an *API* can be invoked by a *conversation*. A *conversation* is any kind of exchange of *messages* (i.e., the conversation uses *messages*). For instance, a *conversation* can be a *call conversation* which usually uses a *request message* and a *response message* (unless the call is a one-way call which omits the response message). Messages have *representations* consisting on multiple *elements*.

Finally, all *operations* are part of the technical *API contract*. This contract usually details all possible *conversations* and *messages* down to the technical *parameter representations* and *addresses*. Thus, the contract describes the *API endpoint*. *API contracts* are necessary for realizing any interoperable and testable technical communication; that is, in order to be able to communicate, *API clients* must comply with the *API provider's contract* for those parts of the *API* that are used. This can be done explicitly at design time (supported by static contracts) or at runtime (requiring dynamic contracts).

These classes and relationships of the domain model form the basic vocabulary for the pattern texts in this paper and related publications. Figure 1 illustrates and summarizes them. Our language, now called Microservice API Patterns (MAP), is organized into categories, two of which are partially published already [23, 31]:

1. *Foundation patterns*: What type of (sub-)systems and components are integrated? Where should an API be accessible from? How should it be documented?
2. *Responsibility patterns*: Which is the architectural role played by each API endpoint and its operations? How do these roles and the resulting responsibilities impact (micro-)service size and granularity?
3. *Structure patterns*: What is an adequate number of representation elements for request and response messages? How are these elements nested? How can they be grouped and annotated (with additional information)? [31].
4. *Quality patterns*: How can an API provider achieve a certain level of quality of the offered API, while using its available resources in a cost-effective way? How can the quality trade-offs be communicated and accounted for? [23].
5. *Identification patterns*: How can API endpoints and operations be found in business requirements and domain models? What is the right approach to service decomposition?

6. *Evolution patterns*: How to deal with life cycle management concerns such as support periods and versioning? How to promote backward compatibility and extensibility? How to communicate breaking changes?

4 PATTERNS FOR THE EVOLUTION OF API DESCRIPTIONS AND THEIR IMPLEMENTATIONS

This paper presents eight patterns that are concerned with the evolution of APIs. An API is not a static standalone product, but part of an open, distributed system. Hence, it has to evolve to adapt to a changing environment: new features are added, bugs and defects get fixed, and some features are discontinued. Our evolution patterns help API owners and designers to introduce API changes in a controlled way, deal with their consequences and manage the impact of these changes on clients depending on them.

Table 1 introduces the patterns via their problem-solution pairs. The relationships between these patterns are shown in a pattern map in Figure 2.

These evolution patterns are directly or indirectly confronted with desired qualities such as:

1. Enhancing compatibility and developer experience.
2. Allowing the provider and the client to follow different life cycles, e.g., a provider can roll out a new API version without breaking existing clients.
3. Minimizing changes to the client forced by API changes.
4. Making it possible for the provider to improve and extend the API and change it to accommodate new requirements.
5. Guaranteeing that API changes do not lead to semantic "misunderstandings" between client and provider.
6. Minimizing the maintenance effort to support old clients.

The different life cycles, deployment frequencies, and schedules of providers and clients of APIs make it necessary to plan API evolution beforehand and forbid making arbitrary changes to an already published API. This problem becomes more severe when the amount of clients using an API increases. On the one hand, the provider's influence on the clients or ability to manage them may shrink if many clients exist (or the clients are not known to the provider). Public APIs are particularly challenging to evolve if alternative providers exist. If no replacement provider is available, on the other hand, API clients depend and rely on the provider to evolve the API (and do so fairly).



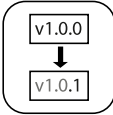
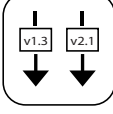

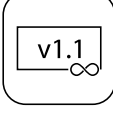
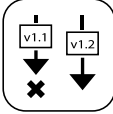
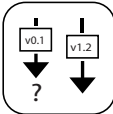
Pattern Icon	Pattern Name	Pattern Summary (Problem and Solution)
	<i>API Description</i>	<p>Problem: Which knowledge should be shared between an API provider and its clients? How should this knowledge be documented?</p> <p>Solution: Define request and response message structures, error/fault reporting, and other technical knowledge to be shared between provider and client. Complement the syntactical interface description with quality management policies, semantic specifications, and organizational information.</p>
	<i>Version Identifier</i>	<p>Problem: How can an API provider indicate its current capabilities as well as the existence of possibly incompatible changes to clients, in order to prevent malfunctioning of clients due to undiscovered interpretation errors?</p> <p>Solution: Introduce an explicit version number into the exchanged messages to indicate the API version.</p>
	<i>Semantic Versioning</i>	<p>Problem: How can stakeholders compare API versions to immediately detect whether they are compatible?</p> <p>Solution: Introduce a hierarchical three-number versioning scheme 'x.y.z', which allows API providers to denote different levels of changes in a compound identifier.</p>
	<i>Two in Production</i>	<p>Problem: How can a provider gradually update an API without breaking existing clients, but also without having to maintain a large number of API versions in production?</p> <p>Solution: Deploy and support two versions of an API endpoint and its operations that provide variations of the same functionality, but do not have to be compatible with each other. Update and decommission (i.e., deprecate and remove) the versions in a rolling, overlapping fashion.</p>
	<i>Limited Lifetime Guarantee</i>	<p>Problem: How can a provider let clients know for how long they can rely on the published version of an API?</p> <p>Solution: As an API provider, guarantee to not break the published API for a given, fixed time-frame.</p>
	<i>Eternal Lifetime Guarantee</i>	<p>Problem: How can a provider support clients that are unable or unwilling to migrate to newer API versions at all?</p> <p>Solution: As an API provider, guarantee to never break or discontinue access to a published API version.</p>
	<i>Aggressive Obsolescence</i>	<p>Problem: How can API providers reduce the effort required to maintain APIs (and their exposed functionality) for existing clients of a previously released API version?</p> <p>Solution: Announce a decommissioning date to be set as early as possible for obsolete API endpoints, operations or message representations.</p>
	<i>Experimental Preview</i>	<p>Problem: How can providers make the introduction of a new API (version) less risky for their clients and also obtain early adopter feedback without having to freeze the API design prematurely?</p> <p>Solution: Provide access to API on a best-effort base without making any commitments about functionality offered, stability, and longevity.</p>

Table 1: Summary of Patterns Presented in this Paper

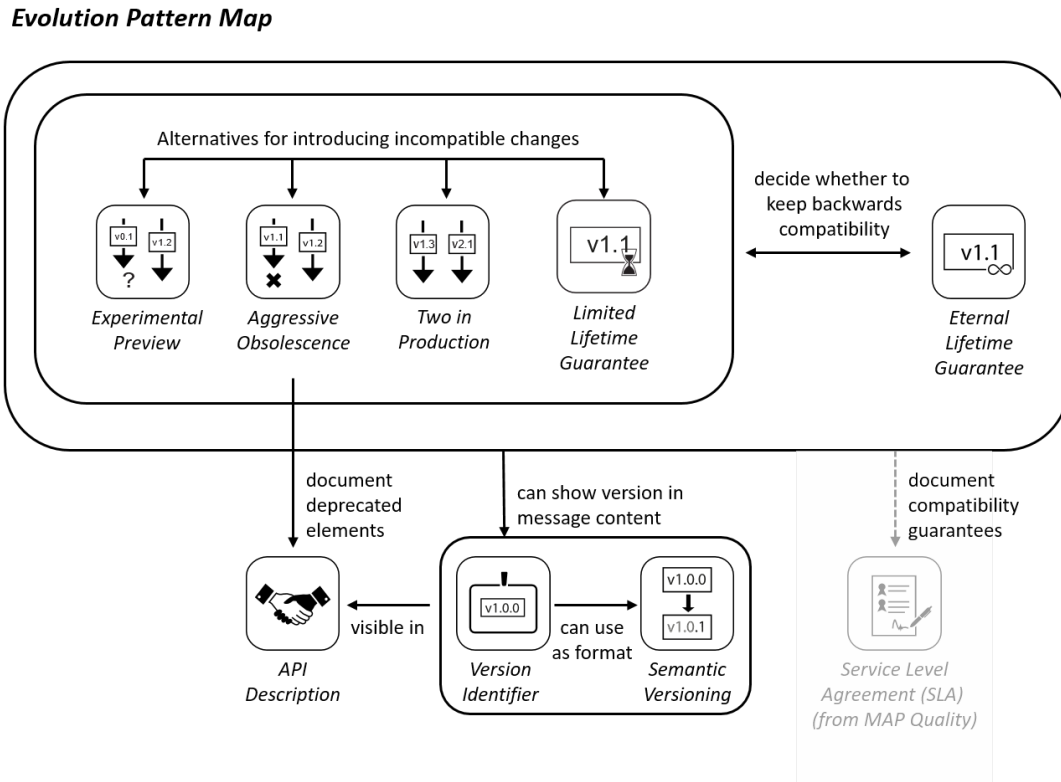


Figure 2: Relationships between the Patterns Presented in this Paper. The SLA Pattern is Documented in [23].

Compatibility and extensibility are typically conflicting quality requirements. Many considerations in the evolution of an API are driven by compatibility considerations. Compatibility is easy to achieve at the time of the initial deployment of the API provider and client: because there is an initially agreed upon version, the client and the provider share the same knowledge; interoperability and accuracy can be designed and tested for. While the API evolves, the shared understanding may begin to vaporize; client and provider drift apart.

Compatibility is a property of the relation between a provider and a client. The two parties are compatible if they can conduct their message exchange and correctly interpret and process all messages according to the semantics of the respective API version. For instance, the provider of API Version n and clients written for this version are compatible by definition (assuming the interoperability tests have passed). If the client for API Version n is compatible with the API provider for version $n - 1$, the provider is *forward-compatible* with the new client version. If the client for API Version n is compatible with the provider Version $n + 1$ the provider is *backwards-compatible* with the old API version.

Compatibility considerations become important as soon as the life cycle of all API providers and all clients cannot be synchronized anymore. With the move of many applications to cloud computing, the number of remote clients has increased significantly (and client-provider relationships keep on changing dynamically). In

modern architecture paradigms such as microservices, an important characteristic is the ability to scale independently, i.e., to run multiple instances of a service at the same time, and also to deploy of new versions with zero downtime. The latter is achieved by having multiple (micro-)service instances running at the same time and switching one service instance to the new version after another until all instances have been upgraded. At least during such transition time, multiple versions of the API are offered. This means that when designing the API and guaranteeing its compatibility, the possibility of having multiple client versions interact with multiple API versions must be taken into account.

The evolution patterns presented in this paper are concerned with conscious decisions about the level of commitment and life cycle support, and keeping or breaking compatibility under different circumstances.



4.1 Pattern: API Description

a.k.a. API Documentation, Explicit and Enhanced Service Contract

Context. A service provider has decided to expose one or more API operations in an API endpoint; the number, name, and synopsis of these API calls have not been specified yet. Therefore, developers of clients (i.e., Web and mobile app developers implementing

Frontend Integrations or the system integrators writing adapters for *Backend Integrations*) are not yet able to code service invocations and do not know what to expect in responses. Furthermore, supplemental interface descriptions are missing as well, including informal explanations of the meaning of the API calls (e.g., parameters in message representations, effects on application state in the API implementation) and related qualities (e.g., idempotency, transactionality).

Problem. Which knowledge should be shared between an API provider and its clients? How should this knowledge be documented? More precisely:

- How can API client and API provider make their agreement on the functional aspects of service invocation (e.g., data transfer representations and invocation prerequisites) explicit?
- How can this functional information be amended with other technical specification elements (e.g., protocol headers, security policies, fault records) and business-level documentation (e.g., call semantics, API owner, billing information, support procedures, versioning)?

Forces. High-level forces to be resolved and balanced when defining shared knowledge in distributed systems include:

- Information hiding (of implementation details)
- Interoperability (of clients and providers written for, and running on, different middleware platforms)
- Consumability (including learnability and simplicity)
- Extensibility and evolvability (as facets of general modifiability and maintainability)

Non-solution. One could only provide basic information such as network addresses and examples of API calls and responses, and many public APIs do just this. Such an approach leaves room for interpretation and is a source of interoperability problems. It offloads work of the API team on the provider side because less information has to be updated during service evolution and maintenance. This comes at the expense of creating extra learning, experimentation, development, and testing effort on the client side.

Solution. Create an *API Description* that defines request and response message structures, error reporting, and other relevant parts of the technical knowledge to be shared between provider and client. In addition to static and structural information, also cover dynamic or behavioral aspects including invocation sequences, pre- and post-conditions, and invariants. Complement the syntactical interface description with quality management policies as well as semantic specifications and organizational information.

How it works. Make the *API Description* both human- and machine-readable. Specify it either in plain text or in a more formal language, depending on the supported usage and integration scenario as well as the maturity of the development practices. Make sure that the semantic specification is business-aligned, but also technically accurate; it must unveil the supported business capabilities² in domain terms so that it is understandable for business analysts (a.k.a. domain subject matter experts), but

²<https://searchmicroservices.techtarget.com/definition/business-capability>

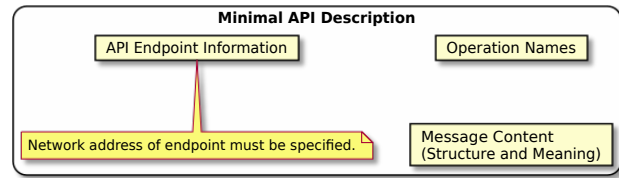


Figure 3: Minimal API Description Variant

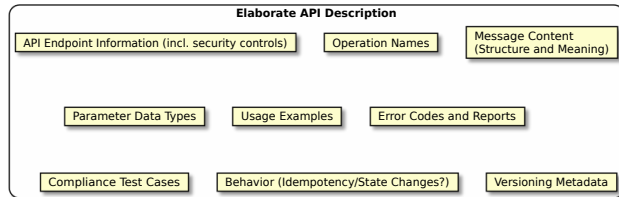


Figure 4: Elaborate API Description Variant

also cover data management concerns such as consistency, freshness, and idempotency. Cover licensing and terms and conditions or factor out this information and define a *Service Level Agreement* (e.g., for business- and mission-critical APIs).

Consider the usage of a recognized functional contract description language such as Open API Specification (f.k.a. Swagger) for the technical contract part of HTTP resource APIs.³

Variants. During pattern mining, we found two variants, *Minimal Description* and *Elaborate Description*. They represent opposite end of a spectrum; hybrid forms can be observed in practice as well.

Minimal Description. At a minimum, clients need to know the API endpoint addresses, the operation names, and the structure and meaning of the request and response message representations. This minimal description forms the technical API contract. In HTTP resource APIs, the operation names are constrained by the HTTP verbs/methods (with the usage of these verbs being defined implicitly/by convention); together with the data contract, they still have to be specified explicitly.

Elaborate Description. More elaborate *API Descriptions* add usage examples, feature detailed tables explaining parameter meanings, data types and constraints, enumerate error codes and error structures in responses, and may even include test cases to check provider compliance. See Chapter 1 as well as Recipes 3.14 and 14.1 in the “RESTful Web Services Cookbook” [1] for related advice.

Example. The “Template for Elaborate API Descriptions” in Figure 5 covers both business information as well as functional-technical API design concerns.

While the template fits on one presentation slide or document page, *API Descriptions* that follow this template probably will require more space. In practice, such API descriptions are often made available via developer portals, project wikis, or service documentation websites.

³Open API Specification (OAS) Version 3.0 has an attribute to share license information.

Service Contract: [Name]	
Business domain (scenario viewpoint, functional area): • [...]	User stories and quality attributes (design forces): • [...]
Service quick reference (synopsis of what this service provides to consumers): • [...]	
Invocation syntax (functional contract); IDL specification, security policy; request/response data samples; endpoint addresses (test deployment, production instances); sample service consumer program (source code); error handling information (error codes, exceptions) • [...]	
Invocation semantics (behavioral contract): informal description of preconditions, postconditions, invariants, parameter meanings; FSM; service composition example; integration test cases • [...]	
Service Level Agreement (SLA) with Service Level Objectives (SLO); Quality-of-Service policies • [...]	
Accounting information (service pricing); external dependencies/resource needs • [...]	
Lifecycle information: current and previous version(s); limitations, future roadmap; service owner with contact information, link to support and bug tracking system • [...]	

Figure 5: Template for Elaborate API Descriptions

Implementation hints. API descriptions in (micro-)service-based systems benefit from some governance and quality control:⁴

- Define an *API product owner* that steers and leads the architectural decision making⁵ for API design and its implementation and decides on the service evolution strategy including versioning. A related pattern is Open Service Ownership⁶ by S. Newman.
- Define the upstream and downstream contract relationship e.g. in the form of one of the relationship types in DDD-style *Context Maps*, for instance open host service and customer-supplier, first described in [8], later picked up by the microservices community, and supported in tools such as Context Mapper⁷.
- Consider to specify Finite State Machines (FSMs) if the API causes non-trivial, possibly long-running state changes. Design the system transaction boundaries carefully; discuss and challenge whether strict or eventual consistency is needed if multiple system parts and clients work with the data that is exposed in the API (e.g., master data, transactional data).
- Distinguish team-internal, floating API *specifications* from longer-lasting API *documentation* [25]. . Make sure that the description is kept up to date as the API and its implementation(s) evolve and are refactored [29]. The API implementation must always match its description; this should not be taken for granted but verified during testing.
- Make sure the service interface and its implementation(s) comply with ethical principles, both general ones and software-specific ones. For instance, see keynote presentations by M. Fowler such as “not just code monkeys”^{8,9}

⁴This can be managed locally, e.g. via technical or organizational stories on agile projects, as opposed to a central design authority.

⁵https://en.wikipedia.org/wiki/Architectural_decision

⁶<http://samnewman.io/patterns/organizational/open-service-ownership/>

⁷<https://contextmapper.github.io/>

⁸<https://www.infoq.com/presentations/healthy-social-environment>

⁹Note that this hint is not API-specific, but applies to all code and other software engineering artifacts produced.

Consequences.

- + A Minimal Description is compact and easy to evolve and maintain.
- + Elaborate Descriptions are expressive. They promote interoperability.
- A Minimal Description might cause client developers to guess or reverse engineer provider-side behavior; such implicit assumptions cause the information hiding principle to be violated and sometimes become invalid in the long run. Furthermore, ambiguities may harm interoperability; testing and maintenance effort increase if new versions that are not backward-compatible are not indicated as such.
- Elaborate Descriptions might introduce inconsistencies due to their intrinsic redundancy where the same elements are mentioned in different parts of the specification. If disclosing provider-side implementation details such as downstream (outbound) dependencies, they violate the information hiding principle. They cause maintenance effort when evolving, primarily the need to systematically update the descriptions (and then consistently implement the changes).

Further discussion. The amount of effort required for *API Descriptions* that meet the information need of the clients depends on the chosen specification depth and level of detail that is required to make meaningful and correct communication possible. If a contract is over-specified, it is hard to consume and maintain (and considered anti-lean because it is seen as unnecessary documentation that qualifies as waste to be eliminated). If it is underspecified, it is easy to read and update, but might still not lead to interoperable client-server conversations that also produce the desired results at runtime. Missing information has to be guessed, assumed, or simply reverse-engineered – e.g., on server-side effects of calls (state changes, data accuracy and consistency), on erroneous input handling, on security enforcement policies, and so on – with no guarantees by the provider on the correctness of the assumptions made by the client. One might consider explaining Quality-of-Service (QoS) policies, e.g. regarding availability, in an explicit *Service Level Agreement* [23].

Whereas the need for informal API descriptions is widely acknowledged, the need for machine-readable technical API contracts (that can be used to generate proxy and stub code, as well as documentation texts) has been discussed controversially. The success of notations such as JSON API and Swagger and tools such as the Apigee console and API Management Gateways suggests that there still is a need for machine-readable technical API contracts in most (if not any) integration scenarios. Many REST books and articles admit that there always is a contract – sometimes called the uniform contract [7]. It just looks different and is created and maintained by different stakeholders.

It is subject to debate whether contracts are really negotiated and agreed upon in practice, or simply dictated by the API provider. Business contexts and API usage scenarios differ: a small startup or a thesis project team consuming the cloud APIs of one of the dominating cloud providers has little hope to be able to request features or negotiate terms and conditions. On the other end of the spectrum, large software vendors and corporate users with Enterprise Level Agreements (ELAs) do exactly this in their strategic

outsourcing deals and cloud partnerships, for instance when rolling out multi-tenant, business-critical applications. Market dynamics and development culture will determine the amount of effort invested in scope and quality of an *API Description*. Client developers can (and should) consider the accuracy and usability of these descriptions in the decision making process when selecting an API and its provider.

Known Uses. The Microservices Canvas¹⁰ template proposed by C. Richardson creates *Elaborate Descriptions* when filled out completely. The template includes implementation information, service invocation relationships, and events produced/subscribed to.

Many concrete examples of *API Descriptions* exist. Here we include Terravis, which uses annotated WSDL according to [16]; so does the Dynamic Interface described in [4]. Developer portals and Do-It-Yourself (DIY) service registries/repositories make these and complementing business/quality policy information available to API users (e.g., frontend and integration developers). A Swiss software vendor specializing on the insurance industry defines its own API documentation format in its internal REST API Design Guidelines.

Technical API description notations exist in many forms for various distributed computing and middleware platforms:¹¹

- Swagger¹², which evolved into the Open API Specification¹³, WADL¹⁴, and RAML¹⁵ for RESTful HTTP. Swagger, for instance, has the notion of a *schema object* to “define input and output data types”¹⁶.
- An alternative is the API Blueprint specification¹⁷ which also has tool support.
- JSON API¹⁸ and APIs.json¹⁹ also suggest contract notations. Note that JSON API²⁰ only covers response message structure and content.
- WSDL 1.1 is commonly used to describe SOAP-based Web services; WSDL 2.0 is more expressive but has seen less adoption in practice.
- The .proto file syntax²¹ used in Google’s Protocol Buffers can be seen as a known use of the (technical/syntactical) API contract concept.
- The Apache Thrift Interface Definition Language (IDL)²² serves a similar purpose and can also be seen as an example of a technical API contract format (data and protocol).
- The Apache Avro IDL²³ originating from the Hadoop project is another example.

¹⁰<http://chrisrichardson.net/post/microservices/general/2019/02/27/microservice-canvas.html>

¹¹Also see this historic overview: <http://restlet.com/company/blog/2017/04/26/a-short-history-of-oai-and-api-specifications/>.

¹²<https://swagger.io/>

¹³<https://www.openapis.org/>

¹⁴<https://www.w3.org/Submission/wadl/>

¹⁵<https://raml.org/>

¹⁶<https://swagger.io/specification/>

¹⁷<https://apiblueprint.org/documentation/specification.html>

¹⁸<http://jsonapi.org/format/>

¹⁹<http://apisjson.org/>

²⁰<https://jsonapi.org/>

²¹<https://developers.google.com/protocol-buffers/docs/proto3>

²²<http://thrift.apache.org/docs/idl>

²³<http://avro.apache.org/docs/current/idl.html>

- AsyncAPI²⁴ allows the creation of machine-readable definitions of “Message-Driven APIs”.

On a broader level, the following notations and assets also qualify as known uses of this pattern:

- Unified Service Description Language (USDL) is a more comprehensive approach that adds metadata and semantic information to the contract, see this W3C report²⁵.
- The SOYA framework²⁶ implements the SOAP Service Description Language (SSDL) specification in C# and on top of Windows Communication Foundation (WCF).
- In the finance industry, SWIFT uses XML (and WSDL/SOAP) as technical interface description language. An example is Alliance Access/Entry 7.0²⁷. SWIFT also standardizes message exchanges²⁸ and has elaborate SLAs, starting with a master SLA²⁹ refined by those for individual services.

SOA maturity levels³⁰ can help with the decision between minimal and elaborate descriptions, supported by the notations and assets listed above.

Related Patterns. Depending on mission criticality and market dynamics, an *API Description* be completed with a *Service Level Agreement* [23] to specify quality goals – and the consequences of not meeting them. Version information and evolution strategies can be included (see for instance *Version Identifier* and *Two in Production* patterns).

Service Descriptor in [6] and *Interface Description* in [26] cover the technical part of the *API Description*.

Other Sources. An online *API Stylebook* collects and references related documentation advice in a dedicated design topic³¹. Recipe 14.1 in the *RESTful Web Services Cookbook* by [1] discusses how to document RESTful Web services. The Engagement Perspective of [32] collects WSDL and SOAP best practices; much of the given advice also applies to other API contract syntaxes. See for instance an OOPSLA 2008 tutorial³². The Australian Digital Transformation Office (DTO) has a best practice rule “document your API”³³ that lists examples of good API documentation.

The notion of Design-by-Contract was established by B. Meyer in [17] in the context of object-oriented software engineering; his advice can also be adopted when defining remote API contracts.

The specific role of data in interface contracts is explained in “Data on the Outside vs. Data on the Inside” by P. Helland [12].



4.2 Pattern: Version Identifier

a.k.a. Explicit Versioning

²⁴<https://www.asyncapi.com/>

²⁵<https://www.w3.org/2005/Incubator/usdl/XGR-usdl-20111027/#L18067>

²⁶<http://soya.sourceforge.net/getting-started.html>

²⁷https://www.swift.com/sites/default/files/resources/swift_functional_overview_alliance_access_entry_7_0.pdf

²⁸<https://www.swift.com/your-needs/iso-20022>

²⁹<https://www.swift.com/about-us/legal>

³⁰<https://www.ibm.com/developerworks/library/ws-soa-method2/>

³¹<http://apistylebook.com/design/topics/documentation>

³²<http://soadecisions.org/download/oopsla08-zimmermann-tut12Handouts.pdf>

³³https://apiguide.readthedocs.io/en/latest/build_and_publish/documentation.html

Context. An API that runs in production evolves. New versions with improved functionality are offered over time. At some point in time, the changes of a new version are no longer backwards compatible, thereby requiring to break existing clients.

Problem. How can an API provider indicate its current capabilities as well as the existence of possibly incompatible changes to clients, in order to prevent malfunctioning of clients due to undiscovered interpretation errors?

Forces.

- *Accuracy and exact identification* of API version
- Minimizing *impact on the client side* caused by API changes
- Guaranteeing that API changes do not lead to *accidentally breaking compability* between client and provider on the semantic level
- *Traceability* of API versions in use for *governance*

Non-solution. Very often, organizations roll out APIs without any idea on how to manage their life cycles. They think that this can be added “later on”. However, lack of governance and versioning has been among the dominant factors that caused past many SOA initiatives and projects to fail [14].

Solution. Introduce an explicit version indicator. Include this *Version Identifier* in the *API Description* and in the exchanged messages. To do so, add a *Metadata Element* to the endpoint address, the protocol header, or the message payload.

How it works. The explicit *Version Identifier* often takes a numeric value to indicate evolution progress and maturity. It can be included in XML namespaces, dedicated version attributes/elements, attribute/element name suffixes, parts of the endpoint URL, the API domain name, or the HTTP content type header. To avoid consistency issues, the *Version Identifier* should appear in one and only one place in all message exchanges supported by an API, unless clients or middleware strongly desire to see it in several places.

To mint identifiers, for example, the three-part *Semantic Versioning* policy can be followed. By referring to this *Version Identifier*, communication parties can check whether they understand and correctly interpret the message.

By indicating a new version with a different *Version Identifier*, the receiving party can abort the interpretation of the message before any further problems occur and report an incompatibility error (in an *Error Report*). The *API Description* can reference features that were introduced in a particular point in time (i.e., the release of a certain version) or that are only available in certain API versions and have been decommissioned later, e.g., when using the *Aggressive Obsolescence* pattern.

Note that API evolution and implementation evolution are two things, as the implementation can evolve separately from the interface (and more frequently be updated). This might lead to usage of multiple version identifiers, one for the remote interface and one for each implementation of this interface.³⁴

³⁴All implementation dependencies and architectural layers should be included in the versioning concept (and/or made sure these dependencies are backward compatible): if the database that implements stateful API calls has a schema that cannot be evolved as fast as the API itself, this might slow down the release frequency. It must be clear which of the two (or more) API versions in production use which version of the backend

Example. In REST APIs the version of different features can be indicated as follows. The version of specific representation formats supported by the client in the content type negotiation headers of HTTP:

```
GET /customers/1234
Accept: text/json+customer; version=1.0
...
```

The version of specific operations is found as part of the the resource identifiers:

```
GET v2/customers/1234
...
```

The version of the whole API can also be specified in the host domain name:

```
GET /customers/1234
Host: v2.api.service.com
...
```

In SOAP/XML-based APIs, the version is usually indicated as part of the namespace of the top-level message element:

```
<soap:Envelope>
  <soap:Body>
    <ns:MyMessage xmlns:ns="http://www.nnn.org/ns/1.0/">
      ...
    </ns:MyMessage>
  </soap:Body>
</soap:Envelope>
```

Another possibility is to specify the version in the payload as in the following JSON example. In the initial version 1.0 of the billing API, the prices were defined in Euro:

```
{
  "version": "1.0",
  "products": [
    {
      "productId": "ABC123",
      "quantity": 5;
      "price": 5.00;
    }
  ]
}
```

With a new version the requirement of multi-currency was realized. This leads to the new data structure and the new contents of the version element:

```
{
  "version": "2.0",
  "products": [
    {
      "productId": "ABC123",
      "quantity": 5;
      "price": 5.00;
      "currency": "USD"
    }
  ]
}
```

(and other dependencies). A “roll forward” strategy or adding a facade that decouples implementation versioning from API versioning may be considered.

If no *Version Identifier* or any other mechanism to indicate a breaking change had been used, old software interpreting the version 2.0 of the message would assume that the product costs 5 EUR although it costs 5 USD. This is because a new attribute has changed the semantics of an existing one. Passing the version in the content type as shown above can eliminate this problem. While it would be possible to introduce a new field `priceWithCurrency` to avoid this problem, such changes lead to technical debt by aggregating over time, especially in less trivial examples.

Implementation hints.

- Stick to a standardized and consistent versioning strategy, e.g., decide which objects to version consistently (operations, data types etc.) or which versioning schema to use (e.g., *Semantic Versioning*).
- When using the *Version Identifier* pattern, take care that client code does not break unnecessarily, i.e., by indicating a new incompatible version although the changes are backwards-compatible. Consider following the *Semantic Versioning* pattern to do so.
- Exposing only the major version of an API that is only used for indicating breaking changes (e.g., Version 3) may be better than exposing the full version (Version 3.1.0). If the latter is visible to the clients, the clients should use the full version information only for information purposes, but not evaluate it or base data validations on it (which is hard to prevent).
- Acknowledge that the versioning needs and rhythms in *Frontend Integration* and *Backend Integration* may be different. For instance, Web and mobile frontends developed by agile teams may change several times per project iteration; these changes may impact also the user stories and corresponding API capabilities to be exposed by the backend. Backends that exchange rarely changing data occasionally, on the other hand, may change their provided and consumed interfaces only rarely, for instance once per calendar year.
- The business sector and market position of the API provider may also have an impact on the change dynamics and API versioning needs. In Domain-Driven Design (DDD) terms [8], an *open host service* exposed by a government organization or monopolist may change rarely, whereas the APIs of public cloud providers that battle for market leadership in *customer-supplier* relationships might change monthly or quarterly.

Consequences.

Resolution of forces.

- + Usage of this pattern helps identifying APIs and communicating about API, operations, and messages clearly.
- + Reduces likeliness of problems due to undetected semantic changes between API versions by breaking compatibility by accident.
- + Makes it possible to trace which message payload version is actually used by clients
- By changing the version identifier, clients might be required to update to a new API version although the functionality that they rely on has not changed; this increases the effort for some API clients.

Further discussion. Applying the *Version Identifier* pattern by itself does not support the decoupling of the provider and client life cycle, but is a prerequisite and enabler for other patterns achieving this. For example, *Two in Production* and *Aggressive Obsolescence* can best be implemented when versions are explicitly signaled.

This pattern describes a simple but effective mechanism to indicate breaking changes, especially those changes that a *Tolerant Reader* would be able to syntactically parse but would fail to interpret the semantics correctly. By making the version explicit, providers can force the client to reject a message of a newer version or can refuse to process an outdated request. This provides a mechanism to safely make incompatible changes. However, it forces the clients to migrate to a new, supported API version. Patterns like *Two in Production* can provide a grace period in which clients can migrate to a new API version.

Using a *Version Identifier* can lead to unnecessary change requests for software components such as the API client. This may happen if the code needs to be changed whenever the API version is changed. When using XML with primitive code generation (e.g., JAXB³⁵), this can be a problem because a change in the namespace will result in a change in the package name, which will affect all generated classes and references in the code to those classes. At a minimum, the code generation should be customized (or more stable mechanisms to access the data be used).

Different integration technologies offer different mechanisms for versioning and have different corresponding practices that are accepted by the respective communities. If SOAP is used, versions are usually indicated by different namespaces of the exchanged SOAP messages although some APIs use a version suffix to the top-level message element. In contrast, parts of the REST community condemn the use of *Version Identifiers* and others encourage the use of HTTP accept and content-type headers³⁶ to convey the version. However, in practice many applications also use *Version Identifiers* in the exchanged JSON/XML or the URL to indicate the version.

When opting for explicit versioning, it must be decided on what level the versioning takes place: In a WSDL, the whole contract can be versioned (e.g., by changing the namespace) or its individual operations (e.g., by having a version suffix). HTTP resource APIs can also be versioned differently: For example, content types, URLs and version elements in the payload can be used to indicate the version. Using smaller units of versioning, e.g. single operations, decreases coupling between provider and client: Consumers might only use features of an API that is not impacted by a change. Instead of bundling operations individually per client (Interface Segregation Principle), fine-grained versioning (on operation or message representation element level) can limit the impact. However, the more elements of an API are versioned, the higher the governance effort is. Both provider and client organizations need to keep track of the many versioned elements and their active versions. Offering APIs specialized for special clients or different client types might be a better design choice in such circumstances.

Known Uses. This pattern has many known uses, both public ones and company- or community-internal ones:

³⁵https://en.wikipedia.org/wiki/Java_Architecture_for_XML_Binding

³⁶http://blog.steveklabnik.com/posts/2011-07-03-nobody-understands-rest-or-http#i_want_my_api_to_be_versioned

- Most public Web APIs use a simple, unstructured *Version Identifier* in the request URI, and many of them add it to response headers as well. A single number is often sufficient. Examples include the Facebook Graph API³⁷ and the Twitter APIs³⁸.
- GitHub uses the HTTP Accept Header³⁹.
- A large European bank places *Version Identifiers* on the operation level in its WSDL contracts and transmits them as part of the request and response messages.
- The Dynamic Interface described in [4] applies the pattern, indicating version identifiers in the service names (i.e., WSDL port types).
- The SOAP-based eCH APIs⁴⁰ use XML namespaces to realize the pattern. An example of such namespace is: `http://www.ech.ch/xmlns/eCH-0134/1`. XML namespaces are commonly defined with URIs.

Related Patterns. A *Version Identifier* can be seen as a special type of *Id Element* and/or *Metadata Element*; “Enterprise Integration Patterns” [13] also discuss more general patterns that are related. The *Version Identifier* can be further structured, e.g., by using the *Semantic Versioning* pattern. The lifecycle pattern *Two in Production* requires explicit versioning; the other life cycle patterns (*Aggressive Obsolescence*, *Experimental Preview*, *Limited Lifetime Guarantee*, *Eternal Lifetime Guarantee*) may also use it.

Applying the *Tolerant Reader* pattern from “Service Design Patterns” [6] allows clients to tolerate some changes.

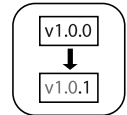
The visibility and role of an API drive related design decisions. For instance, the different life cycles, deployment frequencies, and release dates of provider(s) and client(s) in a *Public API* scenario for *Frontend Integration* make it necessary to plan service evolution beforehand and usually do not allow providers to make arbitrary ad hoc changes to already published APIs due to the impact of such changes on clients (e.g., downtimes, test and migration effort caused). Some or all of these clients might not even be known. A *Community API* providing *Backend Integration* capabilities between a few stable communication parties that are maintained on the same release cycle (and share a common roadmap) might be able to employ more relaxed versioning tactics. Finally, a *Solution-Internal API* connecting a mobile app frontend with a single backend (*Vertical Integration*) owned, developed and operated by the same agile team might fall back to an ad hoc, opportunistic approach that relies on frequent, automated unit/integration tests within a continuous integration and delivery practice.

Other Sources. Because versioning is an important aspect of API and service design, there is much discussion about it in different development communities. The strategies differ widely, and versioning strategies are passionately debated. Opinions range from no explicit versioning at all (sometimes just called “versioning”) because an API should always be backwards-compatible⁴¹ to the different versioning strategies⁴² compared by M. Little.

Chapter 11 of “SOA in Practice” [15] introduces a service life cycle in the context of Service-Oriented Architecture (SOA) design; Chapter 12 discusses versioning.

Chapter 13 of “Build APIs you won’t hate” [24] discusses seven options for versioning (with the *Version Identifier* in URLs being one of these options) and their advantages and disadvantages. It also gives implementation hints.

Chapter 15 of “SOA with REST” [7] deals with Service Versioning for REST.



4.3 Pattern: Semantic Versioning

a.k.a. Version Number Triplet, Three-Number Version

Context. When publishing *Version Identifiers* (dynamically via API payloads) or featuring versioning in the *API Description*, it is not necessarily clear from a single number how significant the changes between different versions are. As a consequence, the impact of these changes is unknown and has to be analyzed by every client. Consumers would like to know the impact to plan the migration to the new versions beforehand and without investing much effort into their own analysis. In addition, providers must manage different versions and thus have to know whether the planned API interface and implementation changes are compatible or break client functionality in order to fulfill any guarantees made to clients.

Problem. How can stakeholders compare API versions to immediately detect whether they are compatible?

Forces. In the context of change management and API versioning, the following forces apply, which must be addressed by the *Semantic Versioning* pattern:

- Minimal effort to detect *version incompatibility* (especially for the client).
- Manageability of API versions and related *governance effort* (e.g., approval processes, quality gates, number of parallel versions, number of branches of API versions).
- Clarity of *change impact*.
- Clear *separation of changes* with different levels of impact and compatibility.
- Clarity with regard to *evolution timeline* of the API.

Non-solution. When marking a new version of an API – regardless of whether using an *Version Identifier* or version numbers elsewhere, the easiest solution is to use simple numbers as versions, e.g., Version 1 followed by Version 2 etc.

However, this versioning scheme does not indicate which versions are compatible to each other (e.g., Version 1 and 3 are compatible, but Version 2 is a new development branch and will be further developed in Version 4 and 5). Thus, branching APIs – for example in a *Two in Production* case – with a plain, single-number versioning scheme is hard because an invisible compatibility graph and several API branches have to be followed. This is because a single version represents the chronological order of releases, but does not address any other concerns.

³⁷<https://developers.facebook.com/docs/graph-api/>

³⁸<https://developer.twitter.com/en/docs.html>

³⁹<https://developer.github.com/v3/versions>

⁴⁰<https://www.ech.ch/de/standards/overviewlist>

⁴¹<https://www.infoq.com/articles/roy-fielding-on-versioning>

⁴²<https://www.infoq.com/news/2013/12/api-versioning#anch104680>

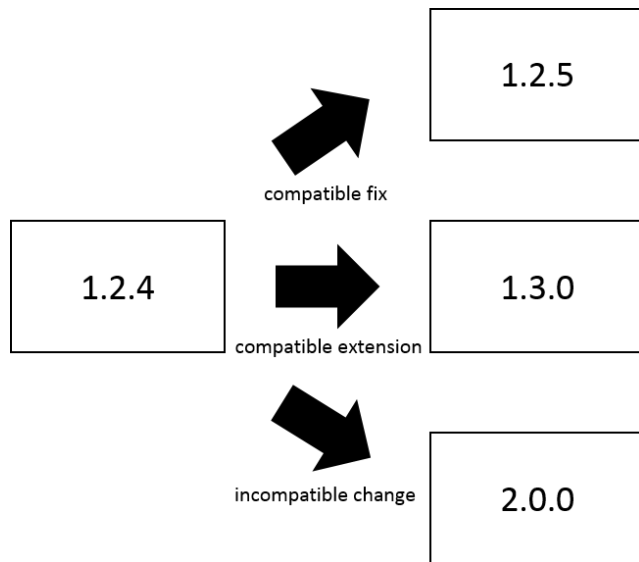


Figure 6: Version Number Changes

Another option is to use the commit IDs of the API revision to be used as the version identifier (depending on the source control system this might not be numeric, e.g., in Git.) While this frees API designers and developers from having to manually assign version numbers, not every commit ID is deployed and no indication of branches and compatibility might be visible to the API client.

Solution. Introduce a hierarchical three-number versioning scheme $x.y.z$, which allows API providers to denote different levels of changes in a compound identifier. The three numbers are usually called *major*, *minor*, and *patch* version.

How it works. A common numbering scheme is illustrated in Figure 6:

- Major version. This number is incremented for incompatible, i.e., breaking changes, e.g., removing an existing operation. For example, a breaking change to version 1.2.4 will result in a new version 2.0.0.
- Minor version. This number is incremented if a new version provides new functionality in a compatible manner, e.g., adding a new operation to an API or adding a new feature to an existing operation. For example, a compatible extension to version 1.2.4 will result in a new version 1.3.0.
- Patch version (also called Fix Version). This number is incremented for compatible bug fixes, e.g., changing and clarifying the documentation in an API contract or changing the API implementation to fix a logic error. For example, a compatible bug fix to version 1.2.4 will result in a new version 1.2.5.

Please note that *Semantic Versioning* only describes how version identifiers are constructed, not where these identifiers are used. This remark applies both to the versioned object (e.g., whole API, individual operations, and data types) and to the places where identifiers are visible (e.g., namespace, attribute contents, and attribute names). *Semantic Versioning* can be applied both to versions that are not communicated to clients and to those that are.

A simplified variant uses only two numbers (or hides the third one from clients, but still uses it internally on the provider side).

Example. A start-up wants to establish itself as a stock exchange data provider in the market. In its first API version (called Version 1.0.0) the API offers a search operation, which can search for a substring of the stock symbol and returns the list of matching stocks including their full names and their prices in USD. Upon customer feedback the start-up decides to offer a historic search function. The existing search operation is extended to optionally accept a time range to provide access to the price historical records. If no time range is supplied, the existing search logic is executed and the last known quote is returned. This version is fully backwards compatible with the old version, i.e., old clients can call this operation and interpret its results. Thus, this version is called Version 1.1.0. Due to a bug in the search, not all stocks containing the supplied search string were found but only those starting with this string. The API implementation is fixed and rolled out as Version 1.1.1 international customers are attracted to the serviced offered by the start-up and request the support of international stock exchanges. As such the response is extended and a currency attribute is now supplied back. This change is incompatible for the client and thus the version is called Version 2.0.0.

Please note that this example is technology-independent on purpose. The supplied data can be transferred in any format, e.g., as JSON or XML objects, and operations can be implemented using any integration technology because this pattern deals with the conceptual problem of issuing version numbers based on the type of change introduced to the API interface or implementation.

Implementation hints.

- Always use a version control system like Git for all your API artifacts. This helps to keep track of versions and artifacts belonging to each other. A version control system also supports branches of versions and makes it easier to see differences between versions.⁴³ The externally visible *Semantic Versioning* identifiers can tag API contract and releases explicitly in the version control system.
- Carefully review the changes to the new API version and decide what type of change to make – and consequently which new version number to assign.
- Explicitly define which changes will lead to an increment of which part of the versioning scheme. Make this classification public and use it as a review checklist when releasing new versions. Apply it consistently and sustainably.
- For assessing the compatibility of changes, a well-designed regression test suite is highly instrumental. If all test cases written for the old version still pass without any change when run against the new version, chances are high that no clients will break. This obviously depends on the quality of the test suite. The better the test suite, the more you can rely on it when assessing compatibility.
- Try to make sure that clients do not interpret nor evaluate the semantic version in an unintended way.⁴⁴ For instance,

⁴³Nowadays, there is no excuse to not use a version control system because they are easy to set up locally and also available as cloud services.

⁴⁴For example, a project returned the full version (including patch version) in its messages, explicitly stating that the contents is for informational and debug purposes

they should not implement version-specific behavior based on the patch version number.

- When creating a new version, review all supporting artifacts such as *API Description* and *Service Level Agreement* for required updates. Schedule these updates and make them part of the definition of done⁴⁵.
- Consider hiding the patch version if using a *Version Identifier*. If a patch version is visible in an attribute or namespace of the exchanged messages, this might by itself accidentally break compatibility and must therefore be avoided. The same can be the case for the minor version, depending on validation rules of the exchanged messages. Especially when using namespaces or attribute names, changes to the exposed version can break compatibility.

Consequences.

Resolution of forces. This pattern resolves its forces in the following ways:

- + High clarity with regards to expressing the impact on compatibility of changes between two API versions
- Increased effort to determine version identifiers because sometimes it is hard to decide to what category a change belongs to

Further discussion. Clear separation of breaking and non-breaking changes is achieved by the semantics of major vs. minor/patch version numbers.

Additional aspects include manageability of API versions and related governance effort. The pattern contributes to resolving this rather broad and cross-cutting force (extra effort is required to fully resolve it).

Transparency of change impact is achieved: the significance and impact of the changes contained in a new version are indicated.

If the pattern is not applied consistently, breaking changes might sneak into minor updates; such process/practice violations should be watched out for and discussed in daily standups, code reviews, etc.

Known Uses. The patterns is in widespread use, in the context of remote APIs and other software artifacts:

- A large Swiss finance institution uses *Semantic Versioning* for expressing the compatibility of their internal services. Major and minor version are exposed via the namespace while the fix version is only part of the contract documentation and is a time-stamp of the latest change.
- Terravis [3] uses *Semantic Versioning* in its service contracts to all parties in order to express compatibility of its external services: major and minor version are exposed via the namespace, while the fix version is only part of the contract documentation. Compatibility between minor versions is achieved by transforming incoming and outgoing messages. This achieves compatibility in the case of minor versions,

and that clients must not interpret this element. However, after incrementing the fix version an important software deployed at many sites broke because of ignoring this processing rule. You want to avoid such situations at all costs because then every little change will break clients.

⁴⁵<https://www.agilealliance.org/glossary/definition-of-done/>

which would otherwise be broken due to a changed namespace. Furthermore it is possible to introduce changes that also restructure message contents (e.g. renaming fields or adding a new hierarchy) as long as the same information and the underlying conceptual model remains the same. The transformation can re-arrange the data accordingly.

- The eBay RESTful API⁴⁶ uses three-number semantic versioning and combines this with *Aggressive Obsolescence*: a new major version remains compatible with an old one, but can deprecate functionality which in the future will be removed. This allows for an easier migration from the point of view of a client although the full version number is visible. However, eBay expects the clients to then migrate within the next six months to the new major version and remove the dependency on deprecated functionality.

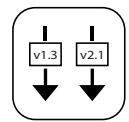
Related Patterns. *Semantic Versioning* requires a *Version Identifier*. *API Description* and/or *Service Level Agreement* can carry the versioning information that matters to clients.

All life cycle patterns are related to this pattern (*Eternal Lifetime Guarantee*, *Limited Lifetime Guarantee*, *Two in Production*, *Aggressive Obsolescence*, and *Experimental Preview*). These patterns differ in the level of commitment given by the API provider; semantic versioning helps to distinguish past, present, and future versions.

In order to improve compatibility between versions, especially minor ones, the *Tolerant Reader* pattern can be used, see [6].

Other Sources. For additional information on how to use semantic versioning in REST, the “REST Cookbook” includes a chapter on versioning⁴⁷. The “API Stylebook” also covers governance⁴⁸ and versioning⁴⁹.

More information on implementing *Semantic Versioning* can be found online at *Semantic Versioning 2.0.0*⁵⁰.



4.4 Pattern: Two in Production

a.k.a. Parallel Versions, Rolling Update Policy

Context. An API evolves and new versions with improved functionality are offered regularly. At some point in time, the changes of a new version are not backwards compatible anymore, thereby “breaking” existing clients. However, clients, especially of a *Public API* or a *Community API*, evolve at different speeds; some of them cannot be forced to upgrade to the latest version in a short time frame because the release cycles of the provider(s) and client(s) do not align well with each other.

Problem. How can a provider gradually update an API without breaking existing clients, but also without having to maintain a large number of API versions in production?

⁴⁶<http://developer.ebay.com/devzone/rest/ebay-rest/content/versioning.html>

⁴⁷<http://restcookbook.com/Basics/versioning/>

⁴⁸<http://apistylebook.com/design/topics/governance>

⁴⁹<http://apistylebook.com/design/topics/versioning>

⁵⁰<http://semver.org/>

Forces. *Two in Production* balances the following forces:

- Allowing the provider and the client to follow *different life cycles* so that a provider can roll out a new API version without breaking clients using the previous API version.
- Guaranteeing that API changes do not lead to *undetected backwards-compatibility problems* between clients and the provider.
- Ability to *roll back* if a new API version is designed badly.
- Minimizing *changes to the client* caused by API changes.
- Minimizing the *maintenance effort* for supporting clients relying on old API versions.

Non-solution. An obvious “solution” is not to care about versioning, always roll out the newest version, and force clients to “live at the head”. For instance, in the past some enterprise-wide Service-Oriented Architecture (SOA) initiatives have been rolled out without any considerations to versioning and life cycle management with the intention to add such capabilities later on. This is difficult, if not impossible: as soon as an API or service is published, clients will use it and are thus bound to the API; not all changes can be expected to be backward compatible. In such situations, organizations have to mitigate the problems as quickly as possible and roll out the new version side-by-side to the old version. While this solves the problem in the short term, the growing numbers of available API versions are hard to maintain for the providers and confusion starts with regard to which API version and associated deployment are to be used.

Solution. Deploy and support two versions of an API endpoint and its operations that provide variations of the same functionality, but do not have to be compatible with each other. Update and decommission (i.e., deprecate and remove) the versions in a rolling, overlapping fashion.

As a variant, consider to support more than two versions, for instance three.

How it works. Such a rolling dual support strategy can be realized in the following way:

- Choose how to identify a version (e.g., by using the *Version Identifier* pattern).
- Offer a fixed number (usually two, thus the pattern name) of API versions in parallel and inform your clients about this life cycle choice.
- When releasing a new API version, retire the oldest one that still runs in production and inform remaining clients (if any) about their migration options. Redirect calls to the retired version, for instance by leveraging protocol-level capabilities such as those in HTTP.

By following these steps, a sliding window of active versions is created (see Figure 7). Thereby, providers allow clients to choose the time of migration to a newer version. If a new version is released, the client can continue to use the previous version and migrate later on (e.g., within a given time frame if the *Limited Lifetime Guarantee* is applied as well). They can learn about the API changes and required client-side modifications without risking the stability of their own primary production system.

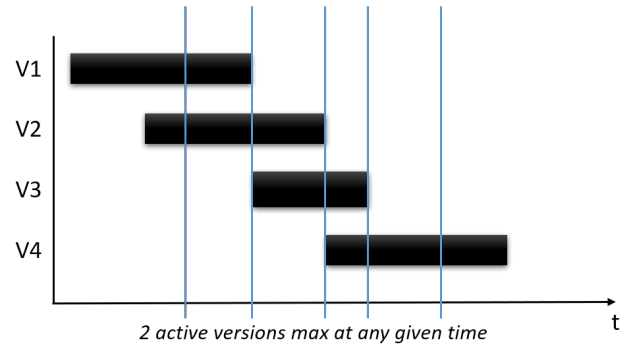


Figure 7: Version Life Cycles when Using Two In Production

Although typically two versions are offered in parallel, this pattern can also be applied in a slightly changed variant as *N in Production* (or *Several in Production*) where the sliding window of active versions is increased to N (with N greater than 2). This strategy gives clients more time to migrate but places more maintenance effort and operational costs on the provider side.

For example, with three in production, one version may be for legacy clients, one for current clients and one for future ones/early adopters (with the newest version serving as *Experimental Preview*). Only when all legacy traffic disappears, Version 1 can be switched off; Version 2 becomes the legacy as Version 3 stabilizes and a new Version 4 can be experimented with. The oldest one then becomes the legacy in turn, and the newest one can be offered to experimental clients while also providing access to the current stable version.

Example. A business software vendor releases an API (Version 1) to its Enterprise Resource Planning (ERP) system. In the continued development of the ERP system, an existing payroll API in the HR module is extended; at some point, new pension plan management features break the API because the data retention policies are incompatible. Because the vendor uses the *Two in Production* pattern, it releases its software with the old API (Version 1) and the new API (Version 2). Customers that use Version 1 can roll out the ERP system (HR module) and then start migrating to use Version 2. New customers in the affected country can start to use the new features of the API Version 2 right away.

With the next release, the software vendor again releases a new API (Version 3) and removes support for Version 1 because Versions 2 and 3 are now supported. Customers that still use Version 1 are cut off until they have migrated to a more recent version (that they can be redirected to). Clients using Version 2 can stay on Version 2 until the next API version is released.

Implementation hints. When applying the *Two in Production* pattern, a number of concerns have to be considered:

- Enforce and enact the deletion policy. We have often encountered client/project situations in which old versions eligible for deletion were still being offered. Usually this happens for two reasons: a) laziness and/or b) pressure by clients. In both situations, the deletion policy should be clearly communicated and reviewed. This is important because not enforcing the deletion policy leads to clients ignoring or forgetting

that versions will get unavailable and rely on officially unsupported versions and taking for granted that these will be available forever. The original provider intention of getting rid of old versions then degenerates quickly, and effectively an *Eternal Lifetime Guarantee* is established implicitly and unofficially.

- Choose the window of active versions. Although the pattern name implies a version window of two active versions, it might be better to use the *N in Production* variant to allow for a larger window, e.g., three or four versions. The *window size* should depend on the agility of the clients and the update frequency of the API: it should be chosen so that clients can realistically follow with the updates. For example, an agile project might release a new API version in every fortnightly or weekly iteration, which makes it impossible for non-agile clients with quarterly deployments to follow when only the latest two versions are supported.
- Do not release incompatible versions when it is not necessary. It is easier for the API designer to make breaking changes (e.g., implied by always changing the *Version Identifier*), but this unnecessarily shifts the window of active versions and creates effort on both the provider and the client side for changing the API version. Compatible changes can and should still be made in the “old” API version. Otherwise consider adopting *Semantic Versioning*.
- In the release notes, make the changes between versions explicit (including changes to non-code artifacts such as *Service Level Agreements*).
- Provide migration examples that take the *Two in Production* approach into account (e.g., feature upgrade from Version 1 to Version 3 in the above example).
- (Micro-)services middleware such as API Gateways [21] and SOA middleware like Enterprise Service Buses [19] can implement the Content-Based Router pattern [13] and mediate calls from older clients to a newer API endpoint. This allows providers to save effort by maintaining only the implementation of the newest API version and providing older clients access via light-weight service adapters.

Consequences.

Resolution of forces.

- + Clients can plan changes well in advance and do not have to migrate exactly when the provider releases a new API version
- + Parallel versions offer a high degree of compatibility and the ability to roll back to an older API version
- + Reduces the likelihood of undetected compatibility changes by keeping old clients on old versions for some time
- + Reduces cost for maintenance by being able to reduce technical debt between different versions without considering backwards-compatibility
- Clients have to adapt to incompatible API changes over time
- Limits ability to respond to urgent change requests
- Causes additional costs for operating multiple API versions
- Quicker changes by provider result in shorter time intervals for client migrations

Further discussion. By using *Two in Production*, the life cycle of provider and client are decoupled to a certain extent: clients do not need to synchronously release their software with the provider. Instead, they have a time window in which they can migrate, test, and release their software. However, clients must migrate as they cannot rely on an *Eternal Lifetime Guarantee* for the API. This means that they have to plan and allocate resources for migrating their software.

The provider can use this pattern to make any changes in a new API version because existing clients will stay on the old version until they migrate. This gives the provider more freedom to clean up the API.

When this pattern is used, the effort of providers and clients are balanced: consumers have a defined time window to migrate to a new API version while providers do not have to support an unlimited number of API versions for an undefined amount of time. As a result, this pattern also defines the responsibilities of both parties to plan their life cycle: the provider can introduce new and possibly incompatible versions but has to support multiple versions, whereas the client must migrate to a new version in a limited time but can more freely and flexibly plan its release schedule.

Known Uses. Known uses for this pattern include:

- Terravis [16] offers two major versions in parallel for two years.
- A large bank in Europe keeps two major versions (see *Semantic Versioning* for definition of “major”) in parallel.
- The Dynamic Interface described in [4] applies the pattern as well.
- GitHub offers a v3 API, but also the next version in parallel⁵¹.
- Facebook describes a rolling release and update policy as well, which is a defining characteristic of *Two in Production*. See their documentation page “Platform Versioning”⁵².

Related Patterns. The usage of this pattern usually requires the *Version Identifier* pattern in order to distinguish the API versions that are currently active and supported concurrently. Fully compatible versions, e.g., as indicated by the patch version in *Semantic Versioning* can replace active versions without violating the *Two in Production* constraints. This should be reported in the *API Description* and/or the *Service Level Agreement*.

Aggressive Obsolescence can be used to force clients to stop using the older API version and migrate to the newer one so that the provider can introduce an even newer API version. If the client requires more guarantees on the expiration date of the old API version, the *Limited Lifetime Guarantee* pattern might be more applicable. An *Experimental Preview* can be one of the versions in production.

Other Sources. “Managed Evolution” covers life cycle management on a general level, but also dives into API versioning. Section 3.5.4 reports a combined usage of *Semantic Versioning* and *Two in Production*. Three versions are reported to have proven as a good compromise between provider complexity and adaptation pace [18].

⁵¹<https://developer.github.com/v3/versions/>

⁵²<https://developers.facebook.com/docs/apps/versions>

“Challenges and benefits of the microservice architectural style”, a two-part article on the IBM Developer portal [9], recommends this pattern.

4.5 Pattern: *Limited Lifetime Guarantee*



a.k.a. Fixed Lifetime, Predefined Support Time Window

Context. An API has been published and made available to at least one client. The API provider cannot manage or influence the evolution roadmaps of its clients, or the damage (e.g., financial or reputation) caused by forcing clients to change their implementation is considered high. Therefore, the provider does not want to make any breaking changes in the published API, but still wants to improve the API in the future.

Problem. How can a provider let clients know for how long they can rely on the published version of an API?

Forces. *Limited Lifetime Guarantee* must balance the following forces:

- Make client-side changes caused by API changes *plannable*.
- Limit the *maintenance effort* for supporting old clients.

Solution. As an API provider, guarantee to not break the published API for a given, fixed time-frame. Label each released API version with an expiration date.

How it works. The provider promises to keep the API usable for a defined, limited but considerably long time and retires it after that. On the one hand, this keeps the client safe from unwanted negative impact or outages. On the other hand, this sets a well-known deadline in advance that the client can plan for.

The advantage of using a fixed time window instead of the number of active versions like in the *Two in Production* pattern is that no further coordination between the provider and client organization is necessary. When accessing the API, the client already knows when it has to adapt and release a software that is compatible with the current API version.

The *Limited Lifetime Guarantee* Pattern stresses the stability for the client side. However, in contrast to its sibling *Eternal Lifetime Guarantee*, it has a built-in expiration time; outdated versions can be decommissioned seamlessly when this time has come. The provider guarantees that the API will never change in an incompatible manner in a pre-announced time-span. The provider agrees to implement any measure to offer the API in a backwards-compatible manner (regardless on the required effort); however, it reserves the right to break or discontinue the API as soon as the validity period expires.

Typical time frames are multitudes of 6 months (6, 12, 18, or 24 months), which seems to provide a good balance for provider and client needs in practice.

Example. One example for a *Limited Lifetime Guarantee* was the introduction of the International Banking Account Numbers (IBAN) in Europe. The limited lifetime was specified in a 2012 resolution of

the European parliament⁵³ granting a period until 2014 after which the old, national account numbers needed to be replaced by the new standard; the use of IBANs became compulsory after that. This regulatory requirement of course had impact on software systems that have to identify accounts. The services offered by such systems had to issue a *Limited Lifetime Guarantee* for the old operations, which used the old account numbers. This example shows that versioning and evolution strategies are not only decided by the API provider alone, but can also be influenced or even mandated by external forces (such as legislation or industry consortia).

Implementation hints. While applying this pattern is straightforward, the ramifications on the provider development side are significant:

- Management might make promises that cannot be met by the development organization. This is often the case for long guaranteed time windows. Prior to issuing such guarantees, the provider organization must plan on how to achieve them and work out realistic life-spans.
- One danger that increases with longer guarantees is that clients do not act on the given far-away deadline, but prioritize short-term feature development higher. This leads to surprises when the API finally is abandoned by the provider after the deadline – that seemed to be far away – has come. Discussions and pleas might lead to an implicit *Eternal Lifetime Guarantee* in such cases. One well-known example is Microsoft’s support for old Windows versions: end-of-life is communicated long before. However, many organizations fail to make (or even plan) the switch to a newer version and negotiate additional support for the old version.

Consequences.

Resolution of forces.

- + Well plannable due to fixed time windows known well in advance
- Limits ability to respond to urgent change requests
- Forces clients to upgrade at a defined point in time which might conflict with their own roadmap and life cycle
- Cannot deal with abandoned clients, i.e., clients that are still in use but no longer actively developed

Further discussion. This pattern is applicable if the provider can constrain the API evolution to include only backwards-compatible changes during the fixed lifetime guarantee. Over time, effort to do so will increase and the API will build up technical debt by introducing changes in a backwards-compatible way that the client can still interpret. This also increases maintenance costs on the provider side for regression testing and maintaining the API operational, which the provider needs to accept.

While the *Limited Lifetime Guarantee* is usually part of the *Service Level Agreement* between provider and client, it has large implications on the provider. The longer the guarantee is valid, the higher the burden is on the provider development organization: In order to keep the published API stable, the provider will usually first try

⁵³<http://www.europarl.europa.eu/sides/getDoc.do?pubRef=-//EP//TEXT+TA+P7-TA-2012-0037+0+DOC+XML+V0/EN&language=EN#BKMD-9>

to make all changes a in backwards-compatible manner. This usually leads to unclean interfaces with awkward names put in place in order to support older and newer clients. If changes cannot be (efficiently) made to the existing version, a new API version might be developed but must run in parallel to the old version in order to fulfill the guarantee (*Two in Production*). This also means that such a guarantee might hinder progress because it is very expensive for a provider to integrate features into the API in the future.

Because the burden is placed on the provider side, clients profit from APIs that are stable for a long time. Their development organizations can easily plan changes to their software. However, the burden on the provider is not as huge as that caused by an *Eternal Lifetime Guarantee* because the provider has the ability to refactor the API to make it consistent again – but only after a defined time, which then has to be stable for the specified time-span.

Uncleanliness of the API due to backwards-compatibility also affects clients: If they want to use one of the integrated new features, they need to cope with the API as it is. Also the API freeze might inhibit progress and integration of new technologies and features by the provider side, which in turn possibly may also hinder clients.

In some settings providers may want to get rid of clients who do not upgrade when the lifetime guarantee expires. For example, due to mistakes in the API design or progress in the area of cryptography security risks to the whole ecosystem of the provider and all clients might arise. Introducing the *Limited Lifetime Guarantee* pattern offers an institutionalized way of enforcing timely client updates.

Known Uses. The *Limited Lifetime Guarantee* pattern is used by several organizations that have different customer types:

- Facebook offers a two-year guarantee on its core API and SDK for everyone, including anonymous clients. See the documentation page Platform Versioning⁵⁴.
- Google Adwords⁵⁵ offers a 10-month guarantee.
- eBay works with deprecation policies⁵⁶.
- Twitter discontinued support for HTTP Basic Authentication on 31.8.2010. The change was announced on 28.4.2010 together with a dedicated countdown clock website⁵⁷ and the initial deadline was set for June 30. All client apps developers had to switch to OAuth by that time.

Related Patterns. The *Limited Lifetime* pattern mixes properties of the *Eternal Lifetime Guarantee* and *Aggressive Obsolescence* patterns: the API must not change in an incompatible way within the announced time-span. However, the time-span serves as an integrated, implicit deprecation mechanism. After it has elapsed, the provider may make any changes including breaking ones or discontinue the expired API version altogether.

The *API Description* and, if present, a *Service Level Agreement* should indicate the actual expiration date for the API version in order to inform API clients by when they need to take action and upgrade. More lenient approaches, giving the provider more freedom with respect to releasing incompatible updates, are presented in

⁵⁴<https://developers.facebook.com/docs/apps/versions>

⁵⁵<https://developers.google.com/adwords/api/docs/sunset-dates>

⁵⁶<http://developer.ebay.com/devzone/rest/ebay-rest/content/versioning.html>

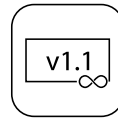
⁵⁷<https://www.programmableweb.com/news/twitter-oauthcalypse-coming-soon/2010/04/28>

the *Aggressive Obsolescence* and *Two in Production* patterns. *Limited Lifetime* and these two patterns can be used together.

A *Limited Lifetime Guarantee* usually has an explicit *Version Identifier*.

Other Sources. Managed Evolution [18] gives rich advice on service versioning and service management processes, for instance including quality gates. Section 3.6 mentions service retirement.

4.6 Pattern: *Eternal Lifetime Guarantee*



a.k.a. Here to Stay, Unlimited Support Period

Context. An API has been made available to at least one client. The integration with its clients has been successful, and these client are used in production. However, one or more of the clients cannot be asked to upgrade to use the latest API version – or cannot be developed further at all.

Problem. How can a provider support clients that are unwilling or unable to migrate to newer API versions at all?

Forces. *Eternal Lifetime Guarantee* must resolve the following forces:

- *No changes* in the client required due to API changes.
- Making it possible for the provider to improve the API and change it to accommodate to *new requirements* of different clients.
- Minimizing the *maintenance effort* to support old clients.
- Ability to *upgrade API infrastructure technologies*.
- Ability to *fix security issues*.
- *Respecting/acknowledging power dynamics* between API provider and client, for instance, the ability of clients to steer API design and evolution.

Non-solution. APIs often are released without a strategy for maintaining and updating them. When an API is already used heavily and a required change poses the question of backwards-compatibility, ad hoc processes might be used to further develop the API. Such ad hoc processes can lead to clients being unable to adapt in time and plan ahead and/or increased unnecessary effort on the provider side. Changes often are postponed because the provider does not want to risk losing part of its user base, thereby implicitly creating *Eternal Lifetime Guarantees* or extending an existing *Limited Lifetime Guarantees* expiration date (of/for old versions) opportunistically whenever an old client asks for that. Hence, having no evolution strategy or following an opportunistic ad hoc approach usually do not work, at least in enterprise settings.

Solution. As an API provider, guarantee to never break or discontinue access to a published API version.

How it works. The API provider promises not to make any breaking changes to the API. This provider-side guarantee reduces the

freedom of the provider to make changes to its API because it is limited to making backwards-compatible changes only.⁵⁸

The guarantee, as any guarantees concerning quality and life cycle aspects, is usually part of a *API Description* and/or *Service Level Agreement* between provider and client. It has large ramifications on the provider.

Example. A national bank wants to offer a service that retrieves a list of ISO currency codes that are used in a country at a given date. Because this API is quite simple and the national bank expects many users to invoke this API, it offers this API with an *Eternal Lifetime Guarantee* so that clients can rely on this service to be available for as long as the national bank can be expected to be open for business (note that only the API contract is fully stable in this example; the returned data and the API implementation might change over time).

Implementation hints. While implementing this pattern by specifying the *Eternal Lifetime Guarantee* is easy, the ramifications on provider development and operations are profound:

- Management or marketing might easily make guarantees that cannot be kept by the development and operations staff. This is especially the case for strong guarantees such as the one expressed by this pattern. It is therefore advisable to plan with the development organization or the DevOps⁵⁹ team on how to achieve this guarantee prior to issuing it.
- The provider should reserve the right to break the guarantee if this is absolutely needed, for instance to be able to close security holes or when the number of clients still invoking the frozen API version drops under a certain threshold.
- Virtualization and/or containerization for instance using Docker can help API providers to reduce the maintenance effort of the implementation by keeping the old production environment (e.g., operating systems, libraries and/or hardware architectures).

Consequences.

Resolution of forces.

- + Clients do not need to change.
- + The provider becomes more attractive as clients can expect the API to remain available for a long time.
- Opportunities for innovation are missed.
- Technical debt is accumulating on the provider side (thus increasing maintenance and operational costs).

Further discussion. By choosing *Eternal Lifetime Guarantee* over other evolution strategies, designers essentially freeze the API at the cost of also freezing innovation and technical progress. This can be useful in extreme scenarios, e.g., where formal software certification is required, but will eventually lead to negotiations about when this guarantee can be broken later on. At the latest, this should happen when the last client has migrated away from the API.

Eternal Lifetime Guarantee is a pattern that maximizes the benefit of stability for clients. It should only be applied if this is the prevalent force because it is counterproductive w.r.t. all other forces: The provider is very constrained in introducing new functionality; it can essentially only make updates on patch level as defined by *Semantic Versioning* thereby remaining completely backwards-compatible. Over time this will result in increased maintenance effort and will either clutter the API design or introduce many parallel active versions as described as the *N in Production* variant of the *Two In Production* pattern. Also API infrastructure technologies cannot be switched easily – if at all – thereby posing both technology risks and security risks: The development technologies used to implement the API interfacing code will eventually go out of support, and it might not be possible to replace broken cryptographic algorithms without breaking backwards-compatibility.

Providing an *Eternal Lifetime Guarantee* usually leads to unclear interfaces with awkward design choices in order to support older and newer clients in parallel. If changes cannot be (efficiently) made to the existing version, a new API version must be developed but has to run in parallel to the old version in order to fulfill the guarantee. This also means that such a guarantee most certainly will hinder progress because it is very expensive for a provider to integrate features into the API in the future.

In case of eternal guarantees, no changes have to be made to keep up with external API dependencies. Clients profit from APIs that they can rely on because their development organizations can focus all further development of the client application on business functionality rather than changes caused by the consumed API.

However, this also has a downside for the client: Because the provider cannot change much it also means it cannot upgrade the service infrastructure technologies and cannot innovate in this API. Therefore, it is likely that technical risks are increasing over time and business value declines or at least does not increase. Because in the long run both the provider and the client will have issues with the frozen API version, it is likely that in a future point in time the life cycle guarantee is changed into a *Limited Lifetime Guarantee*, or the provider even forcefully switches to *Aggressive Obsolescence* or *Two in Production* strategies. Such decision is often motivated by increasing costs that the provider wants to hand over to clients, increasing technical debt on both sides, and missed/reduced opportunities for innovation.

The decision to use *Eternal Lifetime Guarantee* may block disruptive innovations in the foreseeable future because such innovations often cause incompatibilities. One can further question whether things – especially in the digital world – can realistically have an “eternal” lifetime or whether this is just a synonym for a couple of decades.

Known Uses. *Eternal Lifetime Guarantees* is used by a few organizations (including software vendors and alliances) that have different customer types who value sustainability due to the high impact of breaking changes on their own applications and tools:

- SAP used to offer an *Eternal Lifetime Guarantee* (or came close to that) to its paying customers integrating via official and certified Remote Function Calls (RFCs)⁶⁰.

⁵⁸For a discussion of backwards-compatible changes, see the allowed changes for minor or patch fix versions in the *Semantic Versioning* pattern.

⁵⁹<http://bizdevops.uk/>

⁶⁰https://en.wikipedia.org/wiki/Remote_Function_Call

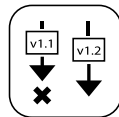
- The local Java APIs for IDE extensions in Eclipse adhere to the API Prime Directive⁶¹ that when “evolving the Component API from release to release, do not break existing clients”. Note that this directive only applies to public APIs. In practice, many API clients also use internal APIs that are prone to incompatibility problems.
- HTTP 1.1 and XML 1.0 are two examples of W3C standards that have remained stable and supported for more than a decade now.

Related Patterns. The guarantee is communicated in the *API Description* and/or in a *Service Level Agreement*.

Compared to other patterns in this evolution category, *Eternal Lifetime Guarantee* is an extreme guarantee that is theoretically unlimited. All other life cycle patterns (*Limited Lifetime Guarantee*, *Two in Production*, *Aggressive Obsolescence*, and *Experimental Preview*) try to limit the guarantee so that there are boundaries for how long backwards-compatibility is to be fulfilled.

An *Eternal Lifetime Guarantee* may have an explicit *Version Identifier* but this is not necessary.

Other Sources. “Managed Evolution” shares information on service governance and versioning. The SLA example in Side-Story 3.5 shows an SLA that states “no phase-out planned” [18].



4.7 Pattern: Aggressive Obsolescence

a.k.a. Early Sunset, Planned Obsolescence (a term discussed in [19])

Context. Once an API has been released, it evolves and new versions with added, removed or changed functionality are offered. In order to reduce effort, API providers do not want to support certain functionalities for clients anymore, e.g., because they are no longer used regularly or are superseded by alternative versions.

Problem. How can API providers reduce the effort required to maintain APIs (and their exposed functionality) for existing clients (of a previously released API version)?

Forces. *Aggressive Obsolescence* needs to balance the following forces:

- Minimizing the *maintenance effort* (e.g., limiting support for old clients)
- Reducing *forced changes to clients* in a given time-span (as a consequence of API changes)
- Acknowledging and respecting *power balances* between API provider and client, for instance, the ability of clients to steer API design and evolution.
- Respecting *commercial goals and constraints*, e.g., impact on *Rate Plan*

Non-solution. One could give no guarantees or follow a rather short *Limited Lifetime Guarantee*, but such weak commitments do not really minimize the impact of changes. One could declare an API to be an *Experimental Preview*, but this is an even weaker commitment that might not be well received by clients.

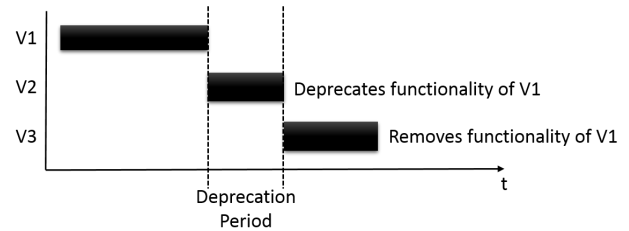


Figure 8: API life cycle when using Aggressive Obsolescence

Solution. Announce a decommissioning date to be set as early as possible for obsolete API endpoints, operations or message representations. Declare such API elements to be immediately deprecated (i.e., still available, but no longer recommended to be used) so that clients have barely enough time to upgrade to a newer or alternative version before the API elements they depend on disappear. Remove the API and the support for it as soon as the deadline has passed.

How it works. When releasing an API the provider should clearly communicate that it follows an *Aggressive Obsolescence* strategy, i.e., that a particular feature might be deprecated and subsequently decommissioned (i.e., removed from support) anytime in the future. When an API, operation or representation element is to be removed, the provider declares this element of the API as deprecated and specifies a time frame by when the feature will be removed completely. Depending on their market position and the availability of alternatives, clients can then choose to upgrade or to switch to a different provider.

Aggressive Obsolescence makes old API versions as a whole – or parts thereof – unavailable rather quickly, for instance within a year for an enterprise application API.

When the needs of the provider(s) outweigh those of the client(s), this *Aggressive Obsolescence* strategy can be followed. By clearly announcing the deprecation and removal schedule for old versions of APIs or API features, the provider can reduce and limit effort for supporting API features that are not support-worthy in a broad sense, e.g., economically because the feature is too costly to maintain (e.g., rarely used features), or legally because a feature becomes unavailable (e.g., introduction of IBAN replaces old account numbers or the introduction of the Euro currency replaces many other currencies). This allows clients to plan the required effort and deployment schedules for still using the service and migrate to an alternative way of achieving the required functionality after a deprecation notice has been issued.

Planning obsolescence and removal usually involves a four-step process (see Figure 8) after a provider releases an API and reserves the right to deprecate and remove parts of it:

0. API version is used in production: Clients happily use the API (Version V1 in the figure)
1. Deprecation: The provider announces the deprecation of an API, API version, or functionality within an API version and indicates when the functionality will be removed (for example, with the next release or at a specific date; Version V2 in the figure)

⁶¹https://wiki.eclipse.org/Evolving_Java-based_APIs#API_Prime_Directive

2. Clients receive the announcement plan and migrate to API replacement versions – or in extreme cases switch to alternative providers.
3. Removal/decommissioning: The provider deploys a new API version that does not support the deprecated functionality anymore. The old version is taken down and retired/archived; requests to the old endpoint can either fail or be redirected to the new version endpoint. (Version V3 in the figure)
4. Clients that did not migrate and depend on removed parts of the API no longer work.

Sometimes *Aggressive Obsolescence* might be the only option for API providers that have not declared any life cycle policies beforehand. If no guarantees are given, deprecating features and announcing – possibly generous – transition periods might be a good way to be able to make incompatible changes again.

Example. A payment provider offers an API that allows clients to instruct payments from their accounts to other accounts. Accounts can be identified by old-fashioned, country-specific account and bank numbers or by International Bank Account Number (IBAN)^{62, 63}. Because IBANs are the new standard and the old account and bank numbers are rarely used, the API provider decides to not support the old numbers anymore. This allows the provider to delete parts of its implementation, thereby reducing the maintenance effort.

In order to allow old clients to migrate to the IBAN scheme, the provider publishes a removal announcement on its API documentation Web page, marks the account and bank number as deprecated in the API documentation, and notifies clients that are registered with the service. The announcement states that the old, country-specific functionality will be removed after one year.

After one year, the payment provider deploys a new implementation of the API that has no support of the old account and bank numbers and removes the old, country-specific attributes from its API documentation. Calls to the old API version fail from now on.

It is notable that in this case, legislature had also specified a transition period to the IBAN system along the lines of deprecating the old, country-specific account number scheme.

Implementation hints. Because *Aggressive Obsolescence* prioritizes the provider side over the client side, it must be taken care that clients are not unrealistically constrained:

- The deprecation and removal schedule should be chosen so that clients can realistically migrate to the new API versions.
- If the clients or a representative subset of clients are known, the deprecation and removal schedule should be communicated to or, even better, negotiated with them.
- In case of anonymous clients the deprecation schedule and the affected parts of the API should be announced early, clearly, and publicly.
- If the same or similar functionality is available elsewhere, the documentation should state how the clients can and should replace the deprecated functionality.

Providers can also offer more advanced help. For example, Facebook offers an API Upgrade Tool⁶⁴ that “will show which of your app’s API calls will be affected by changes in newer versions of the API. This would help mitigate the pain for customers if APIs are aggressively deprecated”.

Consequences.

Resolution of forces.

- + Ideally, clients do not have to change if deprecated functionality is not used.
- + Provider code base is kept small and thus easier to maintain.
- Providers must announce which features are deprecated and when they will be decommissioned.
- Clients that rely on rarely used features or take full advantage of all API features are forced to implement modifications with a schedule which might be unknown when the client is released initially. This schedule usually is communicated to the clients upon deprecation and not during the API release, which might or might not fit the client’s release schedule. Furthermore, it might change later in the API life cycle.
- Clients must get to know obsolete features.

Further discussion. *Aggressive Obsolescence* can be used to enforce a coherent and secure ecosystem around the APIs offered: For example, sunseting weak cryptographic algorithms, out-phased standards or inefficient libraries can help in achieving a better overall experience for all involved parties.

Supporting old clients requires effort for maintaining old API implementations, building service adapters or introducing complicated structures for retaining backwards-compatibility in messages. In order to reduce maintenance effort and technical debt, a mechanism is required that constrains the life-time of functionality and imposes rules that allow existing clients to plan their updates.

The *Aggressive Obsolescence* pattern emphasizes the reduction of effort on the provider side but burdens clients. Essentially, it requires the clients to continuously evolve with the API. In turn, clients stay current with the newest functions and improvements and thus also benefit from switching away from old versions, e.g., they are forced to use new security standards. Depending on the deprecation period, clients can plan and follow API changes like with the *Limited Lifetime Guarantee* pattern but requires them to be rather active.

However, depending on the versioning policy (see *Version Identifier* and *Semantic Versioning*), it is not straightforward to come up with a suitable deprecation and decommissioning policy. If versioning happens on a fine-grained level, e.g., by versioning the content-type of every individual REST resource representation, it is hard to keep track of cluttered and distributedly managed entities for deprecation and removal. Furthermore, communicating the entities to be deprecated and removed is difficult in such scenarios.

Especially in in-house scenarios the knowledge of which systems are using an API (or the deprecated subset of an API) is of great help when deciding if and which features or APIs should be removed. Inter-company services are usually more restrictive and try to better guarantee that other systems might not fail to work properly. Thus,

⁶²https://en.wikipedia.org/wiki/International_Bank_Account_Number

⁶³IBANs originally were developed in Europe, but are now used in other parts of the world as well. They have become an ISO standard.

⁶⁴https://developers.facebook.com/tools/api_versioning/

additional care must be taken before an API or functionality is finally removed. Knowing the relationships between systems and establishing dependency traceability can help with this problem in both scenarios; DevOps practices and supporting tools can be leveraged (e.g., for monitoring and distributed log analysis).

In some business contexts, external clients are managed less diligently than internal clients, and usage dependencies are of no great importance to the API provider. In such circumstances, using the *Rate Plan* pattern (or at least some metering mechanisms) can help identify services to be deprecated and eventually removed. Rate plans can help to financially measure the economic value of an API which can be compared with the maintenance and development effort, thereby deriving an economic decision about prolonging the API lifetime.

Known Uses. The *Aggressive Obsolescence* Pattern is frequently used in large APIs. Some examples are:

- Google has sometimes been reported to implement this pattern in the context of its online services, e.g., Google Reader⁶⁵ and Google Wave⁶⁶.
- Riot Games uses this pattern for the Riot Games API⁶⁷.
- Microsoft uses a 24 month deprecation period on MS Graph⁶⁸.
- Aggressive Obsolescence is also used in local programming APIs, e.g., the ZEND Framework⁶⁹.
- Mike Amundsen reports the use of this pattern in “Microservices in Practice, Part 2” [20].

Related Patterns. Other patterns defining the life cycle also deal with the discontinuation of services. Several strategies can be employed, as outlined in the *Two in Production*, *Limited Lifetime Guarantee* and *Eternal Lifetime Guarantee* patterns. In contrast, our *Aggressive Obsolescence* pattern can be used in a more fine-grained way because it aims at removing functionality from an API, which is not necessarily bound to syntactic units: While other versioning schemes are attached to operations or set of operations, only certain representation elements might get deprecated and removed in *Aggressive Obsolescence*, thereby allowing less obstructive changes.

Another difference to other patterns is that *Aggressive Obsolescence* always uses relative timeframes for removing functionality: Because functionality becomes obsolete during the lifetime of an API, it gets flagged deprecated within its active period and the deprecation period runs from this time onwards. In contrast, *Two in Production* or *Limited Lifetime Guarantee* can be used with absolute timeframes based on the initial release date.

Finally, the pattern cannot only be applied proactively, but also reactively during API maintenance.

The pattern may or may not use a *Version Identifier*. If present, an *API Description* or a *Service Level Agreement* should indicate the usage of this pattern.

⁶⁵<https://www.google.com/reader/about/>

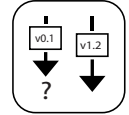
⁶⁶<https://support.google.com/answer/1083134?hl=en>

⁶⁷<https://discussion.developer.riotgames.com/articles/652/riot-games-api-v3.html>

⁶⁸https://developer.microsoft.com/en-us/graph/docs/concepts/versioning_and_support

⁶⁹<https://github.com/zendframework/zend-mvc/commit/0a531e25bc8ec8e10b4730dfb609b6ff34f8b7b6>

Other Sources. “Managed Evolution” [18] shares general information on service governance and versioning, for instance how to define quality gates and how to monitor traffic. Chapter 7 deals with “Measuring the Managed Evolution”.



4.8 Pattern: *Experimental Preview*

a.k.a. Beta Program, Testing Sandbox, API Preview

Context. A provider is developing a new API or a new API version that differs significantly from the published version(s) and is still under intensive development. As a result, the provider wants to be able to freely make any modifications necessary. However, the provider also wants to offer its clients early access so that these clients can start integrating against the new API and comment on the proposed API functionality and structure (in support of an iterative and incremental, or even agile, integration development process).

Problem. How can providers make the introduction of a new API (version) less risky for their clients and also obtain early adopter feedback without having to freeze the API design prematurely?

Forces. The use of this pattern is driven by the following forces:

- Make room for *innovations and new features* (often developed iteratively and incrementally)
- Obtain early *feedback* (for providers)
- *Focus efforts* (in early development, e.g., avoid API governance efforts)
- Offer *early learning opportunities* (for consumers)
- Desire to be able to rely on *API stability* (from a client point of view)

When offering a new API version, or even more so when offering a completely new API, providers want to showcase interim versions during development to their future customers. This raises the awareness of the new API (version), facilitates feedback and gives the customers time to decide whether to use the new API and plan development projects to make use of them. However, providers are still actively developing the API and want to retain the freedom to make arbitrary changes and to rapidly address client feedback.

Non-solution. The provider could just release a new API version when the development is finished. However, this means that clients can only start developing and testing against the API from the release date onwards. Developing the first client implementations might take several months; during this time, the API cannot be used yet, leading to revenue losses (for commercial APIs).

One way to counter these problems is to release API versions frequently. While this allows the client to sneak a peek on an API, the provider has to manage many versions and will probably release many incompatible changes; this increases the governance effort further and makes it challenging for clients to closely track the latest API version.

Solution. Provide access to API on a best-effort base without making any commitments about functionality offered, stability, and

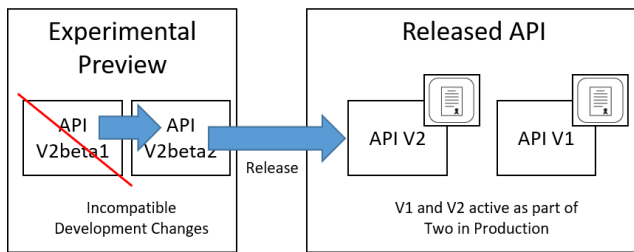


Figure 9: Changes in Experiment Preview Sandbox and Production

longevity. Clearly articulate this lack of API maturity explicitly (to manage client expectations).

How it works. By releasing an unstable version as an *Experimental Preview* in an ungoverned development sandbox, the provider makes an API version available to clients outside of the normal governance process (e.g., not governed by a *Service Level Agreement*, but still documented by an draft or intermediate *API Description*). Consumers can voluntarily opt to test and experiment with the new API version knowing that they cannot rely on its availability, stability or any other quality criteria; by definition, the *Experimental Preview* API might even disappear suddenly or after a pre-announced *Limited Lifetime Guarantee*. Having early access to the API preview is especially beneficial for clients which have to estimate the effort required for integrating with the final version and jump starting the development while the API development is not yet finished.

Figure 9 illustrates the pattern: The *Experimental Preview*, which covers the pre-release guarantees, is complemented by an application of *Two in Production* for governing the life cycle of productive APIs here. The *Experimental Preview* can either be made available to all known or unknown clients; alternatively, a closed user group can be selected for it (to limit support and communication effort).

Learning and helping the provider to try out a new API and its features is different from writing production applications; as a provider, you can introduce a grace period to ease the transition from beta to the production version (see for instance “API Design at GitHub”⁷⁰).

Example. Let us assume that a software development company X wants to create a new product that lets it leave its comfort zone because it goes beyond the functionality offered in existing products: X has been active in the development of a continuous build and deployment solution, currently offered as a cloud software service with a Web-based online user interface. Development teams use the service to build the software by fetching a revision from a repository and deploying the artifacts to configurable servers. Large customers have now requested an API to better trigger and manage builds and receive notifications about build states besides the Web interface. Because X has not yet offered any APIs and thus lacks knowledge and experience, the developers choose an *Experimental Preview* of the API and improve it continuously by incorporating feedback from the customers that decide to adopt it early.

Implementation hints.

- Additional effort is required to maintain and operate a second environment besides the production environment. Depending on how many clients opt to experiment and whether a minimum quality level should be guaranteed, the required resources should not be underestimated.
- Care should be taken to move APIs from the preview stage to production because the APIs will be governed by stricter rules after such move. In the best case, the API has been tested so well (and improved accordingly) that incompatible changes are unlikely in the foreseeable future.
- A “beta for life” approach does not help clients trying to integrate the API because at some point they wish to have a stable API to rely on, ideally also governed by a *Service Level Agreement*, e.g., for performance and availability.
- Even though different API versions are clearly separated, their implementations including databases might be shared. For example, data in the *Experimental Preview* of the API might be consolidated from existing data sources or by sharing data storages. Note that such data sharing can be subject to enterprise-level principles and contextual constraints; as such they might require certain approvals or might be prohibited at all.

Consequences.

Resolution of forces.

- + Clients have early access to innovation and opportunity to influence the API design, thereby living according to agile values⁷¹ and principles⁷² such as welcoming change and responding to it continuously.
- + Providers have the flexibility to freely and rapidly change the API before declaring it stable.
- Providers may find it difficult to attract clients due to the lack of long term commitment to the API perceived as immature.
- Clients have to keep changing their implementation until a stable version is released.
- Clients might face total loss of investment if a stable API is never released and/or the preview disappears suddenly.

Further discussion. By offering an API in a non-production environment that is closely linked to the current development version, providers can offer peek previews into a new API or API version to interested clients. For this environment, different – and usually very lax – service levels (e.g., regarding availability) are guaranteed. Consumers can intentionally decide to use this relatively unstable environment for giving feedback on the new API design and its functionality and to start development. However, clients can also use only the production API (that is still provided with the standard service levels and thus is usually more stable and reliable).

Early clients perform a form of acceptance testing as they might find inconsistencies and missing functionality in this API version resulting in changes without requiring the provider to follow a full-blown governance process.

When applied at the right time and with the right scope, this pattern allows and/or deepens the collaboration between providers and their clients, and enables clients to roll out software that utilizes

⁷⁰<https://events.yandex.com/lib/talks/42/>

⁷¹<http://agilemanifesto.org/>

⁷²<https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/>

new API functionality quicker. However, the provider organization must maintain and run an additional environment with different API endpoints. It also makes its development progress on new APIs more transparent. This includes changes (and mistakes) that are not part of the final API, which become visible to outside partners.

Known Uses.

- GitHub offers API Previews⁷³ following this pattern, for instance for its GraphQL support (at the time of writing).
- Facebook also applies this patterns for its core API. For instance, there is an Audience Network SDK Beta Program⁷⁴.
- Google had a reputation to roll out and get stuck in beta phase for a long time; see for instance this opinion piece⁷⁵.
- Sandboxes and beta programs are commonly used by many cloud service providers and their APIs; for instance the Platform-as-a-Service (PaaS) offering of Swisscom, a Swiss cloud provider, had this status in 2015.

Related Patterns. *Experimental Preview* is similar to traditional beta (testing) programs. It is the weakest support commitment an API provider can give, followed by *Aggressive Obsolescence*. When transitioning the API to a productive environment, another life cycle governance (a.k.a. evolution strategy) pattern must be chosen, e.g., *Two In Production*, *Limited Lifetime Guarantee*, or even (when really required) *Eternal Lifetime Guarantee*. When the *N in Production* variant of *Two In Production* is applied, an *Experimental Preview* can be combined with any of these patterns.

The *Experimental Preview* may have a *Version Identifier* but does not have to.

An *API Description* should clearly state which version is experimentally previewed and which one is productive. Specific *API Keys* can be assigned to grant certain clients access to the preview/the beta version.

Other Sources. For tips and tricks about beta testing, refer to the DZone article “Beta Testing of Your Product: 6 Practical Steps to Follow”⁷⁶.

5 SUMMARY AND OUTLOOK

In this paper we presented eight patterns that can be applied when creating or adapting an evolution strategy for an API.

An early decision in a API project is whether an API and its endpoints should be versioned at all, and, if so, how and on which levels of granularity. These decisions are usually documented in an *API Description*. The *Version Identifier* pattern presents a basic solution for this problem set by explicitly versioning API elements, e.g., by transmitting a version number in the request messages.

If simple numeric version identifiers are insufficient because major and minor changes have to be distinguished from backward-compatible patches, three-number *Semantic Versioning* can be applied for APIs just like it is often done today for code artifacts or entire software products.

⁷³<https://developer.github.com/v3/previews/>

⁷⁴<https://developers.facebook.com/blog/post/2018/10/08/introducing-the-audience-network-sdk-beta-program/>

⁷⁵http://www.slate.com/articles/news_and_politics/recycled/2009/07/why_did_it_take_google_so_long_to_take_gmail_out_of_beta.html

⁷⁶<https://dzone.com/articles/beta-testing-of-your-product-6-practical-steps-to>

Once a versioning scheme is in place and APIs run in production, it has to be decided how many of versions should be supported in parallel. The *Two in Production* pattern defines and limits the currently active API versions in support of a rolling release strategy. Another possibility is to fix the duration of how long individual versions are supported by using the *Limited Lifetime Guarantee* pattern. *Aggressive Obsolescence* can be used to selectively decommission and later remove an API version or a functionality subset. If the provider chooses to make unlimited compatibility guarantees, the *Eternal Lifetime Guarantee* pattern might be considered. Finally, an *Experimental Preview* status can be given to API endpoints or operations (or new versions of them) to avoid premature commitments, but permit sneak previews or beta programs available to all or selected clients.

These patterns have been mined and validated by capturing real-world API documentation and conducting workshops with practitioners. They are part of a larger effort to document Microservice API Patterns (MAP). The already published patterns are publicly available at <https://microservice-api-patterns.org/>.

In the future, we consider to extend our pattern language with additional patterns belonging to other categories in MAP. For instance, additional structural representation patterns as well as patterns concerning the architectural roles and responsibilities of endpoints and operations within an API are currently being mined, captured, and validated. API endpoint and service identification strategies and tactics as well as corresponding artifacts form another candidate pattern category. Context Maps, Bounded Contexts and Aggregates from Domain-Driven Design [8] seem to be particularly promising starting points for microservice API design.

ACKNOWLEDGMENTS

We want to thank the following individuals and groups for their invaluable input and feedback: participants of EuroPLOP 2019 Workshop D, Andrei Furda, Hans-Peter Hoidn and colleagues, Stefan Kapferer, Uwe van Heesch, and Rebecca Wirfs-Brock.

REFERENCES

- [1] Subbu Allamaraju. 2010. *RESTful Web Services Cookbook*. O'Reilly.
- [2] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. 2003. *Web services: Concepts, Architectures and Applications*. Springer.
- [3] Walter Berli, Daniel Lübke, and Werner Möckli. 2014. Terravis – Large Scale Business Process Integration between Public and Private Partners. In *Lecture Notes in Informatics (LNI), Proceedings INFORMATIK 2014*, Erhard Plödereder, Lars Grunke, Eric Schneider, and Dominik Ull (Eds.), Vol. P-232. Gesellschaft für Informatik e.V., Gesellschaft für Informatik e.V., 1075–1090.
- [4] Michael Brandner, Michael Craes, Frank Oellermann, and Olaf Zimmermann. 2004. Web services-oriented architecture in production in the finance industry. *Informatik-Spektrum* 27, 2 (2004), 136–145. <https://doi.org/10.1007/s00287-004-0380-2>
- [5] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. 2007. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. Wiley.
- [6] Robert Daigneau. 2011. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley Professional. <http://www.servicedesignpatterns.com/>
- [7] Thomas Erl, Benjamin Carlyle, Cesare Pautasso, and Raj Balasubramanian. 2013. *SOA with REST - Principles, Patterns and Constraints for Building Enterprise Solutions with REST*. Prentice Hall. 1–XXXII, 1–577 pages.
- [8] Eric Evans. 2003. *Domain-Driven Design: Tackling Complexity In the Heart of Software*. Addison-Wesley.
- [9] André Fachat. 2019. Challenges and benefits of the microservice architectural style. <https://developer.ibm.com/articles/challenges-and-benefits-of-the-microservice-architectural-style-part-2/>
- [10] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schuheck, and Peter Arbitter. 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and*

Manage Cloud Applications. Springer.

- [11] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley.
- [12] Pat Helland. 2005. Data on the Outside Versus Data on the Inside. (2005), 144–153. <http://cidrdb.org/cidr2005/papers/P12.pdf>
- [13] Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.
- [14] Nils Joachim, Daniel Beimbom, and Tim Weitzel. 2013. The influence of SOA governance mechanisms on IT flexibility and service reuse. *The Journal of Strategic Information Systems* 22, 1 (2013), 86–101. <https://doi.org/10.1016/j.jsis.2012.10.003> Service Management and Engineering in Information Systems Research.
- [15] Nicolai Josuttis. 2007. *SOA in Practice: The Art of Distributed System Design*. O'Reilly.
- [16] Daniel Lübke and Tammo van Lessen. 2016. Modeling Test Cases in BPMN for Behavior-Driven Development. *IEEE Software* 33, 5 (Sept.-Oct. 2016), 15–21.
- [17] Bertrand Meyer. 1997. *Object-oriented Software Construction (2nd Ed.)*. Prentice-Hall.
- [18] Stephan Murer, Bruno Bonati, and Frank Furrer. 2010. *Managed Evolution - A Strategy for Very Large Information Systems*. Springer.
- [19] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai M. Josuttis. 2017. Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software* 34, 1 (2017), 91–98. <https://doi.org/10.1109/MS.2017.24>
- [20] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai M. Josuttis. 2017. Microservices in Practice, Part 2: Service Integration and Sustainability. *IEEE Software* 34, 2 (2017), 97–104. <https://doi.org/10.1109/MS.2017.56>
- [21] Chris Richardson. 2016. *Microservice Architecture*. <http://microservices.io>. (2016).
- [22] Chris Richardson. 2018. *Microservices Patterns*. Manning.
- [23] Mirko Stocker, Olaf Zimmermann, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. 2018. Interface Quality Patterns - Crafting and Consuming Message-Based Remote APIs. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLOP '18)*.
- [24] Phil Sturgeon. 2016. *Build APIs you won't hate*. LeanPub, <https://leanpub.com/build-apis-you-wont-hate>.
- [25] Uwe Van Heesch, Theo Theunissen, Olaf Zimmermann, and Uwe Zdun. 2017. Software Specification and Documentation in Continuous Software Development: A Focus Group Report. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLOP '17)*. ACM, New York, NY, USA, Article 35, 13 pages. <https://doi.org/10.1145/3147704.3147742>
- [26] Markus Voelter, Michael Kircher, and Uwe Zdun. 2004. *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. J. Wiley & Sons, Hoboken, NJ, USA.
- [27] Chris Wood, Art Anthony, Arnaud Lauret, and Kristopher Sandoval. 2016. *The API Economy: Disruption and the Business of APIs*. Nordic APIs AB, Stockholm, Sweden. <https://nordicapis.com/api-ebooks/the-api-economy/>
- [28] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. 2018. Guiding Architectural Decision Making on Quality Aspects in Microservice APIs. In *16th International Conference on Service-Oriented Computing ICSOC 2018*. 78–89. <http://eprints.cs.univie.ac.at/5956/>
- [29] Olaf Zimmermann. 2015. Architectural Refactoring: A Task-Centric View on Software Evolution. *IEEE Software* 32, 2 (Mar.-Apr. 2015), 26–29. <https://doi.org/10.1109/MS.2015.37>
- [30] Olaf Zimmermann. 2017. Microservices Tenets. *Comput. Sci.* 32, 3-4 (July 2017), 301–310. <https://doi.org/10.1007/s00450-016-0337-0>
- [31] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. 2017. Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLOP '17)*. ACM, Article 27, 36 pages. <https://doi.org/10.1145/3147704.3147734>
- [32] Olaf Zimmermann, Mark Tomlinson, and Stefan Peuser. 2003. *Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects*. Springer Science & Business Media.