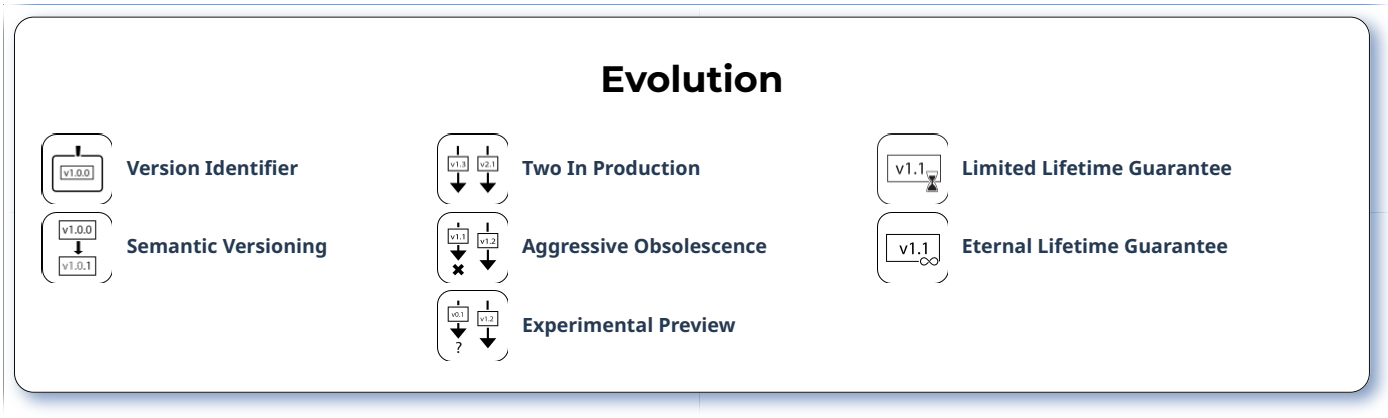
Evolution Patterns

The evolution category deals with lifecycle management concerns such as versioning and release/decommissioning strategies.

← Quality Patterns Foundation Patterns →

Category Overview

The Evolution Patterns category comprises the following patterns:

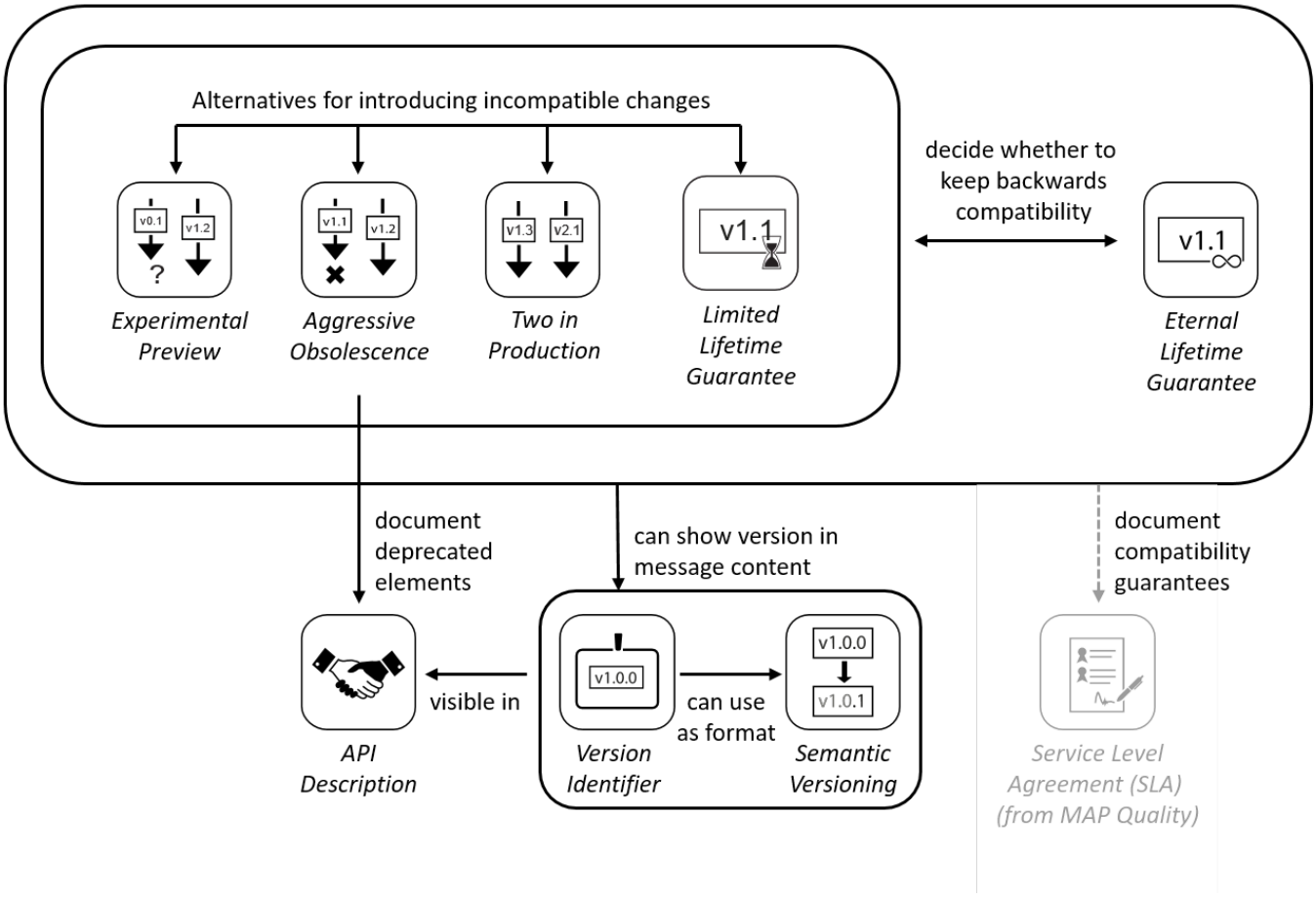


An early decision in a API project is whether an API and its endpoints should be versioned at all, and, if so, how and on which levels of granularity. These decisions are usually documented in an API Description. Version Identifier pattern presents a basic solution for this problem set by explicitly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

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.

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. 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 can be used. 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.

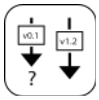
Evolution Pattern Map

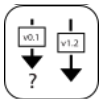


Problem-Solution Pairs

	**PATTERN: AGGRESSIVE OBSOLESCENCE**
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?
Solution	Announce a decommissioning date to be set as early as possible for obsolete API endpoints, operations or representations. Declare such API element 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 once the decommissioning date has arrived.

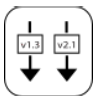
	**PATTERN: ETERNAL LIFETIME GUARANTEE**
Problem	How can a provider support clients that are unable or unwilling to migrate to newer API versions at all?
Solution	As an API provider, guarantee to never break or discontinue access to a published API version.

	**PATTERN: EXPERIMENTAL PREVIEW**
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?

	**PATTERN: EXPERIMENTAL PREVIEW**
Solution	Provide access to API on a best-effort base without making any commitments about functionality offered, stability, and longevity. Clearly articulate this lack of API maturity explicitly (to manage client expectations).

	**PATTERN: LIMITED LIFETIME GUARANTEE**
Problem	How can a provider let clients know for how long they can rely on a published API version?
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.

	**PATTERN: SEMANTIC VERSIONING**
Problem	How can stakeholders compare API versions to immediately detect whether they are compatible?
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.

	**PATTERN: TWO IN PRODUCTION**
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?
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.

	**PATTERN: VERSION IDENTIFIER**
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?
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.

← Quality Patterns

Foundation Patterns →

← BACK TO HOMEPAGE

Microservice API Patterns