# Semantic Version Management (Semantic Versioning)

## Introduction

In modern IT systems, especially in the case of microservice architectures, APIs, and libraries, it is critically important that different components remain compatible with each other. Semantic Versioning (SemVer for short) is a standardized approach that carries information about the nature of changes through version numbers.

The goal of SemVer:

- improving dependency manageability
- reducing compatibility problems
- making integration easier

## Basic Principles

The semantic version format is:

```
MAJOR.MINOR.PATCH
```

where:

| Element | Meaning | When do we increase it? |
|---|---|---|
| MAJOR | major version | in the case of an incompatible change |
| MINOR | minor version | in the case of a new, compatible feature |
| PATCH | patch | in the case of a bug fix |

Each element of the version number is a non-negative integer, and changes in increasing order.

## The logic behind the version number

One of the most important ideas of SemVer is that the version number is a communication tool between developers.

Example:

1.2.3 → stable API

1.2.4 → same API, only a bug fix

1.3.0 → new features, but old code still works

2.0.0 → major version change → old code may break

# Version increment decision process

Increasing the version is not random, but a rule-based decision.

flowchart TD A["A change occurred"] --> B{"Compatible?"} B -- "No" --> C["Increase MAJOR"] B -- "Yes" --> D{"New feature?"} D -- "Yes" --> E["Increase MINOR"] D -- "No" --> F["Increase PATCH"]

# The three version levels in detail

## MAJOR version

Increasing the MAJOR version means that a change occurred in the system that is not backward compatible.

Examples:

- changing the parameter list of a function
- deleting an API endpoint
- modifying a data structure

This is the most critical type of change, because:

- clients may work incorrectly without an update
- integration errors may appear

## MINOR version

Increasing the MINOR version means introducing new features that do not break existing operation.

Examples:

- adding a new API endpoint
- introducing an optional parameter
- new service module

Important:

- old clients continue to work
- the PATCH value is reset to zero in this case

## PATCH version

Increasing the PATCH version serves exclusively for bug fixes.

Examples:

- bug fix
- performance improvement
- documentation clarification (if it does not affect the API)

This is the safest update type.

# Lifecycle of versions

## 0.x.x – development phase

- there is no stable API
- any change can happen
- not recommended for use in a production system

## 1.0.0 – stable release

This is the point where:

- the API can be considered stable
- following the SemVer rules becomes mandatory

# Pre-releases and metadata

## Pre-release

```
1.0.0-alpha
1.0.0-beta.1
1.0.0-rc.1
```

Meaning:

- not yet a stable version
- intended for testing
- lower priority than the final version

## Build metadata

```
1.0.0+build.123
```

Characteristics:

- information only (for example build number)
- does not affect the ordering of versions

# Comparing versions

The ordering of versions happens according to the following logic:

1. MAJOR → MINOR → PATCH
2. then pre-release

Important:

- build metadata does not count
- a pre-release is always “weaker” than a normal version

# Important rules

- A released version cannot be modified
- Every change requires a new version
- Defining the public API is mandatory
- Version numbers increase monotonically

# Practical significance in integration

Semantic versioning is especially important in the following cases:

- communication between microservices
- use of REST / GraphQL APIs
- integration of external libraries
- CI/CD pipelines

Advantages:

- automatic dependency updates safely
- reducing compatibility errors
- better version traceability

# Example of version evolution

```
 1.2.3 → bugfix → 1.2.4 1.2.3 → new feature → 1.3.0 1.2.3 → breaking change → 2.0.0
```

# Summary

Semantic versioning is a simple but extremely effective method for tracking the evolution of software in a structured and understandable way, while minimizing integration risks and increasing system reliability.