How Are Web APIs Versioned in Practice? A Large-Scale Empirical Study

Souhaila Serbout* and Cesare Pautasso

Software Institute (USI), Lugano, Switzerland
E-mail: souhaila.serbout@usi.ch; c.pautasso@ieee.org
**Corresponding Author*

Received 12 September 2023; Accepted 21 March 2024

Abstract

Web APIs form the cornerstone of modern software ecosystems, facilitating seamless data exchange and service integration. Ensuring the compatibility and longevity of these APIs is paramount. This study delves into the intricate realm of API versioning practices, a crucial mechanism for managing API evolution. Exploring an expanded and diverse dataset of 603 293 APIs specifications created during the 2015–2023 timeframe and gathered from four different sources, we examined the adoption of the following versioning practices: Metadata-based, URL-based, Header-based and Dynamic versioning, with one or more versions in production. API developers use more than 50 different version identifier formats to encode information about the changes introduced with respect to the previous version (i.e., semantic versioning), about when the version was released (i.e., age versioning) and about which phase of the API development lifecycle the version belongs (i.e., stable vs. preview releases).

Keywords: API, Web API, OpenAPI, empirical study, versioning.

Journal of Web Engineering, Vol. 23_4, 465–506.
doi: 10.13052/jwe1540-9589.2341

1 Introduction

The continuous evolution of software is an integral aspect that applies also to Web APIs [13, 18, 28]. A particular type of software meant to be integrated and reused in various systems, Web APIs empower developers to build innovative applications by leveraging data and services from various sources. However, as these APIs evolve, developers face a challenge: how to introduce changes without disrupting existing clients depending on them [15, 19, 24]. This challenge underscores the critical importance of API versioning.

API versioning [16] is a fundamental practice that enables API providers to manage change effectively while ensuring compatibility with existing clients. API providers often use version identifiers to make changes evident to clients, allowing them to refer to specific versions of the API on which they depend. In some cases, providers make multiple versions of the same API available to ease the transition for clients as they switch from retired versions to newer versions [21].

The lack of a centralized registry for Web APIs, combined with the flexibility for service providers to use their own versioning approaches [25], has led to multiple and sometimes inconsistent practices in terms of discoverability and notification of breaking changes [11]. While versioning metadata is required when describing Web APIs according to the OpenAPI specification, developers use a variety of version identifier formats to express different concerns: when was the API released, whether the API version is stable or still a preview release, whether the changes introduced in the API are likely to break clients. Such variability in versioning practices raises questions about the prevalence of semantic versioning [1] adoption among Web APIs and how to dynamically discover and select which API version is available at runtime.

While in our preliminary study [27] we focused on a single source of API specifications, in this paper we have significantly broadened our OpenAPI description dataset (from 186 259 to 602 859 specifications) to mitigate the external threads to validity. In addition to GitHub, we mined SwaggerHub and APIs.guru, some of the preeminent platforms for API specification sharing. We have also fine-tuned our parser to detect 21 additional version identifier formats (257 as opposed to 236 in [16]).

More in detail, we have also delved into the adoption of header-based versioning practice, which involves including version information within HTTP headers rather than in the API URL or metadata. A practice which we did not cover in [16]. We observed a remarkable diversity in this practice, with developers employing 126 distinct header names to convey versioning information.

More in detail, we aim to answer the following research questions:

Q1: What are the commonly adopted practices for Web APIs versioning?

Q2: How do developers distinguish stable from preview releases?

Q3: To what extent is the practice of semantic versioning adopted in Web APIs, and are there alternative versioning schemes in use?

Q4: What is the prevalence of APIs with multiple versions in production? how many concurrent versions exist?

Q5: How has the adoption of dynamic versioning and header-based versioning practices evolved over time?

Q6: How sensitive are the results to the source of the API descriptions?

The remainder of this paper is organized as follows: Section 2 provides some background on the topic of Web API versioning. Section 3, describes the methodology used to collect and analyze the four datasets. The following Section 4 presents the analysis results, structured around each versioning practice. More precisely: Section 4.1 discusses the practice of metadata-based versioning, where version identifiers are included within the API metadata. Section 4.2 explores URL-based versioning, where version identifiers are embedded as part of endpoints URLs. Section 4.5 delves into header-based versioning, where the desired version is specified using an HTTP header. Section 4.6 discusses dynamic versioning, where version identifiers are discovered by retrieving them from dedicated API endpoints. Section 4.7 examines the practice of having multiple versions of the same API available in production simultaneously. Section 4.8 presents an analysis of the different version identifier formats used over the years. Section 5 summarizes the findings by answering the research questions, leading to a proposal for a more structured representation of versioning metadata in the next revision of the OpenAPI specification (Section 6). Related work is outlined in Section 7, after which the paper concludes in Section 8 with a summary of the main results and a discussion of future work.

A replication package is available on GitHub [2].

2 Background

2.1 Version Identifiers in Web APIs

In the realm of Web APIs, there exist various options for including a version identifier, statically, as part of an API description, or dynamically, as part of messages exchanged with the API. In this study, we expect to find evidence for the following practices:

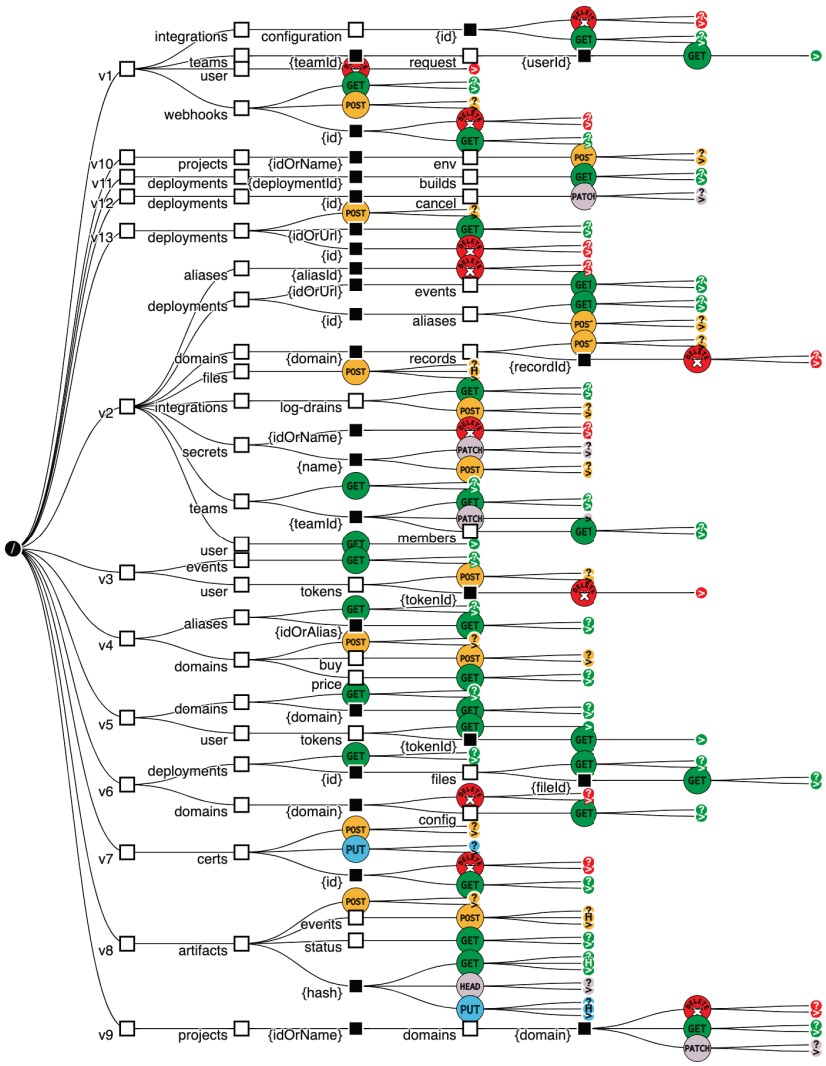

Figure 1 Tree visualization of the structure of a subset of the Vercel API [3]. Different version identifiers (v1-v12) are found in the path URL addresses.

• Metadata-based versioning: The version identifier is included within the API metadata. This can be achieved using industry-standard formats like the OpenAPI Specification, which provides a comprehensive description of the API, including versioning details, in a machine-readable manner or mentioning it on the API provider's website or documentation.

• URL-based versioning: The version identifier expected by the client can be embedded as part of HTTP request messages as a parameter or a segment in the endpoint path URL, such as:

```
https://<server-address>/<path>?<query>
```

where:

```
<server-address> = <version-identifier>.<dns-domain>

<path> = <path>/<version-identifier>/<path> || <path> || ""

<query> = <query>&version=<version-identifier>&<query> || <query> || ""
```

For example:

`https://v1.example.api/path`

`https://api.example.com/v1/path`

`https://api.example.com/?version=v1`

Embedding version identifiers in endpoint URLs is commonly used also when multiple versions of the API coexist simultaneously, known as the "two in-production" pattern [21]. The API server employs the version identifier found within the request to route the request to the appropriate API version. For example, in the Vercel API 12 different versions are accessible to clients (Fig. 1).

• Header-based versioning: Instead of embedding version information in the URL or other parts of the request, when an API uses header-based versioning the client specifies the desired version using an HTTP header. This type of versioning can be also applied at the operation level where the version is specified in the header of the request associated with the operation, such as the example in Listing 1.

Listing 1 Header-based versioning applied to certain API operations

```
name: x-ms-version
in: header
description: The version of the operation to use for this request
    ↪  See https://docs.microsoft.com/en-us/rest/api/
    ↪ storageservices/versioning-for-the-azure-storage-services
    ↪ for details
required: false
schema:
    type: string
```

E.g., within our dataset, the Amaysim[1] API serves as an illustrative example. It uses the `accept-version` header as a mechanism for transmitting the version identifier conforming to Semantic Versioning (SemVer). In the event that no header is explicitly provided, the API defaults to invoking the latest available version:

One of the advantages of header-based versioning is that the clients can seamlessly switch between versions without modifying the request structure. The server interprets the header to route the request to the appropriate version of the API.

• Dynamic versioning: In APIs utilizing header-based versioning, developers must explicitly instruct API consumers on specifying the intended API version in request headers. This information, encompassing the designated header field (e.g., `x-api-version`) and the requisite version format (e.g., `v1`, `1.0`), must be documented. API consumers are then tasked with including the version data in the headers of their HTTP requests using the provided header name and value or the version query parameter. For instance, in the case of the GitHub API, developers are informed about viable version header values through the invocation of a `GET /versions` endpoint. This endpoint facilitates the retrieval of a list encompassing available version identifiers. Developers can thus reference this endpoint to ascertain the valid version options for configuration within their request headers. This practice augments transparency and streamlines the process of selecting and incorporating appropriate API versions.

2.2 OpenAPI Versioning Metadata

API service providers typically provide API clients with information on how to use the API through a description, which is often written in natural language [31] or based on a standard Interface Description Language (IDL), such as OpenAPI [4]. This later has seen a widespread adoption across industries [17, 29, 30], which underscores its pivotal role in modern API development and integration. It is also a form of documentation that is machine-readable, enabling systematic analysis on a large scale.

OpenAPI offer a standardized, language-agnostic framework for documenting RESTful APIs, which facilitates clearer communication among developers, accelerates development timelines, and ensures consistent API

[1] https://www.amaysim.com.au/

implementation. Moreover, it includes a specific required field {`"version": string`} in the `info` section pertaining to the API's metadata. However, there are no constraints on the format used to represent the version identifier. Additionally, version identifiers can be embedded in the API endpoint addresses, which are stored in the `server` and `path` URLs.

While the OpenAPI standard defines how developers describe their APIs, there is no centralized standard documentation manager service where developers can share API specifications. For example, SwaggerHub [5] does not impose any rules on the format of version identifiers, nor does it require developers to upgrade them when publishing a new version of the API description. We aim to study the resulting variety of version identifier formats found in a large collection of OpenAPI descriptions.

2.3 API Stable Releases

API stable releases represent the versions of the API that are deemed ready for use in production environments. These versions have undergone thorough testing and are considered reliable and stable for use by clients. The version identifiers for stable releases often convey important information about the changes introduced in the release, the compatibility with previous versions, and the maturity of the API.

In our study, we identified four primary classes of formats for stable release identifiers:

• *Major Version Number:* This format is characterized by a single integer value that increments with each major release. It is a simplified form of semantic versioning, focusing only on major changes that are likely to be incompatible with previous versions. This format is straightforward and easy to understand, but it does not provide detailed information about minor updates or patches.

• *Semver (Semantic Versioning):* The goal of semantic versioning [1] is to reflect the impact of API changes through the version identifier format `MAJOR.MINOR.PATCH`. The `MAJOR` version counter is incremented when incompatible API changes were introduced, the `MINOR` counter is upgraded when new functionalities were added without breaking any of the old ones, and the `PATCH` increases for backward compatible bug fixes.

Several widely known package managers, such as NPM [6], Maven [32], and PyPI, adopt semantic versioning as a standard for package version identifiers. These package managers enforce the usage of semantic versioning and perform version increment checks every time the package is republished [14].

We put under this category all the version identifiers that follow the semantic versioning format, regardless of the number of counters used, starting from 2 counters.

• *Date:* Some APIs use the release date as the version identifier. This format can take various forms, such as YYYY-MM-DD or YYYYMMDD. It provides a clear timeline of API releases and is easy to understand. However, it does not provide any information about the overall impact of the changes introduced in each version.

• *Tag:* This format uses arbitrary word values as version identifiers, such as: “latest”, “newest”, “test”, which we found as the most common words. This format provides the most flexibility, but it can also be the most difficult to understand and manage, especially for APIs with many versions.

2.4 API Preview Releases

Test releases are often given specific marketing names to clearly reflect their purpose and distinguish them from stable releases. Marketing names help also to indicate the audience of the test releases, and allow users to understand that they should expect bugs [7, 8, 22].

In our datasets, we identified the following six types of usage for preview release tags:

• *Develop:* A version under development is still in the process of being created and is not yet complete or stable. It may contain new features or bug fixes that have not yet been fully tested, and may not be suitable for use in a production environment. Developers may use dev versions to test new features and make changes before releasing a final version to the public.

• *Snapshot:* These versions are automatically built from the latest development code and are intended to be used by developers.

• *Preview:* These are unstable versions that are made available to users before the final release. Preview versions are typically released to a small group of users or testers to gather feedback and iron out any bugs or issues before the final release. They can also be used to give users a preview of new features to expect to see in the next stable version.

• *Alpha:* These versions are considered to be very early in development and are likely to be unstable and contain many bugs. They are often released to a small group of testers for feedback.

• *Beta:* These versions are considered to be more stable than alpha versions and are often released to a wider group of testers for feedback. They may still contain bugs, but they are expected to be closer to the final release.

• *Release Candidate (RC):* These versions are considered to be very close to the final release and are often the last versions to be released before the final version. They are expected to be stable and contain only minor bugs.

Our goal is to quantify how often such types of stable and pre-release versions are found, and whether developers also use other kinds of tags to classify their API versions.

3 Methodology

3.1 Dataset Preparation

Our analysis was performed on OpenAPI specifications which we collected in four datasets: GitHub (5 218 APIs, 165 939 commits); SwaggerHub (387 463 APIs), BigQuery (45 467 APIs), APIs.guru: (3 990 APIs), for a total of 602 859 API descriptions.

- GitHub: This historical dataset of 165 939 OpenAPI specifications, belonging to 5 218 APIs, was extracted from GitHub utilizing its API. Our approach involved systematically querying the contents of JSON and YAML files within the GitHub repositories. Upon detection of an OpenAPI specification file, we embarked on the retrieval of its complete version history, as well as its associated dependencies. Similar to previous works [13], also in this study we have included only APIs with the entire history of valid specifications and at least 10 commits in their version history, thereby filtering trivial or inconsequential repositories.

- SwaggerHub: We assembled an expansive dataset of OpenAPI specifications sourced from SwaggerHub. Our data collection methodology hinged on leveraging both the Swagger Proxy API and Swagger API and tactically implemented strategies to ensure the efficient retrieval of data while adhering to the API's rate limits. Out of the retrieved 432 265 specifications, we could keep 387 463 unique and valid specifications.

- BigQuery: BigQuery is a fully managed, serverless data warehouse and analytics platform that enables fast and scalable querying of large datasets using SQL-like queries. It contains a snapshot of GitHub which is updated on a weekly basis [9]. We extracted OpenAPI files from BigQuery utilizing a distinct methodology from that employed by Assetnote's team [10], who employed a query strategy targeting files named "swagger.json", "openapi.json", and "api-docs.json". This resulted in

17 741 files (running the query the 8th, September 2023). This method could potentially overlook numerous OpenAPI files, owing to the lack of well-defined conventions or stipulations concerning file nomenclature.

Given that an OpenAPI file conforming to version 3.0 or later must invariably include the "openapi" key, or "swagger" if aligning with version 2.0, along with the mandatory "paths" key, our query strategy involved targeting files containing either of these key terms. As a result, we identified a total of 175 549 files, from which 45 467 represented unique and valid OpenAPI specifications.

However using the BigQuery public GitHub dataset, it is not possible to query the history of specific files, since the results do not contain the needed pointers to retrieve the list of files affected by a specific commit in a repository.

- APIs.Guru: This open-source project and community-driven platform aims to provide a comprehensive, curated and up-to-date collection of API specifications. We fetched the OpenAPI files discovered through the APIs.guru API, as follows:

```
wget https://api.apis.guru/v2/list.json ; cat list.json | jq -r
'.[]["versions"][]["swaggerUrl"]' > urls
wget -i urls
```

We found 3992 OpenAPI files, where only two were invalid and each one belongs to a distinct API.

A distinctive feature of the GitHub dataset lies in the comprehensive historical record of API specification commits, complete with their respective timestamps. Conversely, the artifacts in the SwaggerHub collection include metadata such as their creation date and the last modification date of the specifications. In Figure 2, we give an overview of the yearly distribution of APIs commits in the case of GitHub dataset and the number of APIs created every year in the case of SwaggerHub dataset. This will make it possible to track the adoption of API versioning practices over the past years.

The approach of the analysis remains consistent across all the specifications from all sources. Only the datasets obtained from GitHub and SwaggerHub provided the necessary timestamps for the creation of specifications. Utilizing these timestamps enabled a time-series analysis to observe the adoption patterns of dynamic versioning (referenced in section 4.6) and to track the evolution in the adoption of diverse formats over time (detailed in section 4.8).

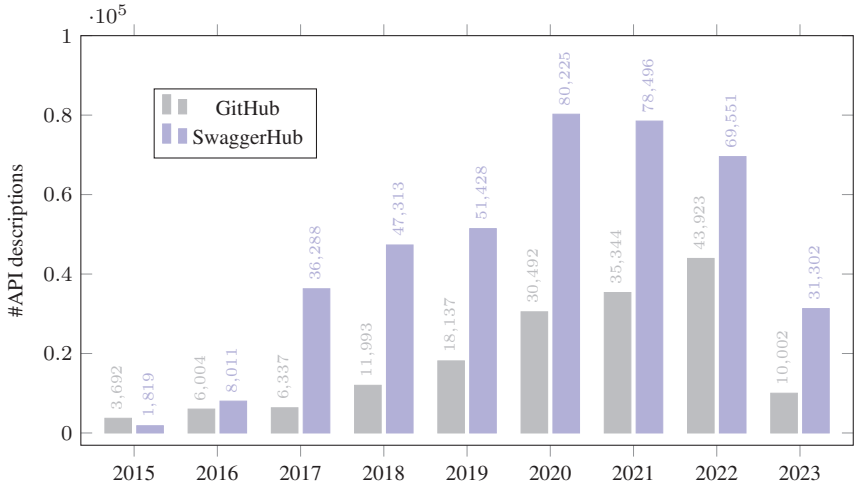

Figure 2 Number of artifacts in the GitHub and SwaggerHub datasets over the years.

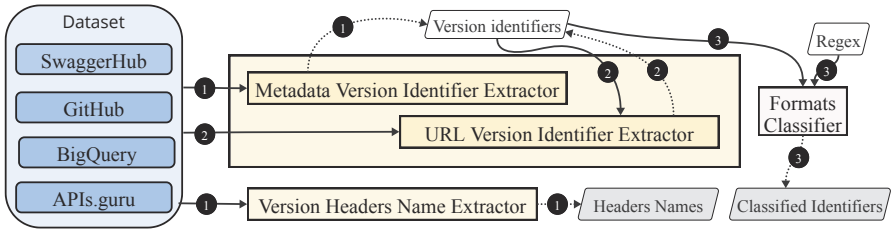

Figure 3 Versioning analysis pipeline.

3.2 Analysis Methodology

To perform this study, we automated the extraction of versioning metadata and the detection of different versioning practices by analyzing 602 859 API specifications written in the OpenAPI description language [4].

As depicted in Figure 3, we first retrieved 10 221 distinct version identifiers from the `info.version` field in each OpenAPI description in each dataset (see the third column of Table 2 for the number of unique version identifiers found in each dataset). We then searched for any of these identifiers in the URL addresses listed as part of the endpoints or server URL strings.

To classify the version identifiers, we employed a set of regular expression rules (Table 1). These detectors were iteratively defined based on our observations to ensure that most of the samples could be labeled. We also distinguished between version identifiers used to describe preview releases

Table 1 Some detectors used to classify the version identifier formats

Format	Regular Expression
integer	`/^(\d{3}|\d{2}|\d{1})+$/i`
v*	`/v\d*/i`
semver-3	`/^(v|)\d+\.\d+\.\d+$/i`
date(yyyy-mm-dd)	`/^\d{4}-\d{2}-\d{2}/`
semver-dev*	`/^(v|)\d+\.\d+(\.\d)*(\.|-)dev\d*$/i`
semver-snapshot*	`/^(v|)\d+\.\d+(\.\d)*(\.|-)SNAPSHOT\d*$/i`
date-preview*	`[date](-|\.)preview$/i`
v*alpha*	`/^v\d+alpha\d*$/i`
v*beta*	`/^v\d+beta\d*$/i`
semver-rc*.*	`/^(v|)\d+\.\d+(\.\d)*-rc\d*\.\d+$/i`

and stable versions of the APIs. The complete list of regular expression rules are included in the replication package.

Given such a variety of sources, we examine the specifications and present the results collectively and individually, based on the origin of the specifications, to determine how the outcomes vary according to their sources.

4 Results

4.1 Metadata-based Versioning

Metadata-based versioning involves embedding the API version identifier within the API documentation itself. In the context of OpenAPI-documented APIs, this approach is facilitated by a designated `info.version` field within the specification. This field empowers developers to explicitly denote the version of the web API being documented. By articulating the version as a string in the OpenAPI specification, the practice of metadata-based versioning establishes a clear means to communicate and represent the API's versioning information.

4.1.1 Metadata-based versioning adoption overview

The `info.version` field, while obligatory for a valid specification, is found to accommodate various values including empty strings and other non-conforming string entries, such as: `""`, `"null"`, `"undefined"`, `"version unknown"`, `"-"`, `"_"`, `"unknown"`, `"VERSION_PLACEHOLDER"`, `"no version"`, etc. These entries were identified and subsequently excluded and considered as no metadata based versioning was used.

Table 2 Number of artifacts featuring metadata-based versioning

Dataset	#APIs	#Unique Version IDs	Most Used ID	#APIs
GitHub	5 107 (97.87%) 162 244 (97.77%) commits	7 020	`1.0.0`	2 242 42 416 commits
BigQuery	44 364 (89.94%)	1 941	`1.0.0`	4 987
SwaggerHub	381 437 (94.30%)	8 616	`1.0.0`	240 310
APIs.guru	3 988 (99.95%)	824	`v1`	275
Total	592 033 (98%)	10 221		

In Table 2, we report that the vast majority of artifacts (across all datasets, more than 90%) makes use of metadata-based versioning. The number of unique version identifiers detected within each dataset is listed in the third column. The most common version identifier is `1.0.0`, while `v1` is the mostly used one only in the APIs.guru, where 7% of the APIs which use metadata-based versioning have the `v1` identifier.

4.1.2 Version identifiers formats

Given that the version is represented as a string, discerning a consistent format for the extracted version identifier becomes a non-trivial task. To address this challenge, we developed a parsing mechanism leveraging 257 regular expressions. This parser enables the detection and classification of diverse version formats employed within the dataset, enhancing our ability to systematically analyze and categorize the extracted version identifiers. In Table 4, we present the top 20 frequently employed version identifier formats observed in each of the four study datasets.

The format `semver-3` was found to be the most frequent format. But, looking at each dataset independently, we can see that the most frequently adopted version identifier format varies depending on the source. For the SwaggerHub dataset, the most common format is `semver-3`, accounting for 69.78% of the total. Similarly, the `semver-3` format is also the most prevalent in the GitHub dataset, representing 61.68% of the total.

In contrast, the BigQuery dataset primarily uses the `date(yyyy-mm-dd)` format, which constitutes 31.82% of the total. The APIs.guru dataset also favors a date-based format, specifically `date(yyyy-mm-dd)`, which accounts for 39.77% of the total.

The `Other` category of formats encompasses all version identifier formats that could not be classified due to their non-uniformity. These formats do not adhere to any of the common versioning schemes such as Semantic

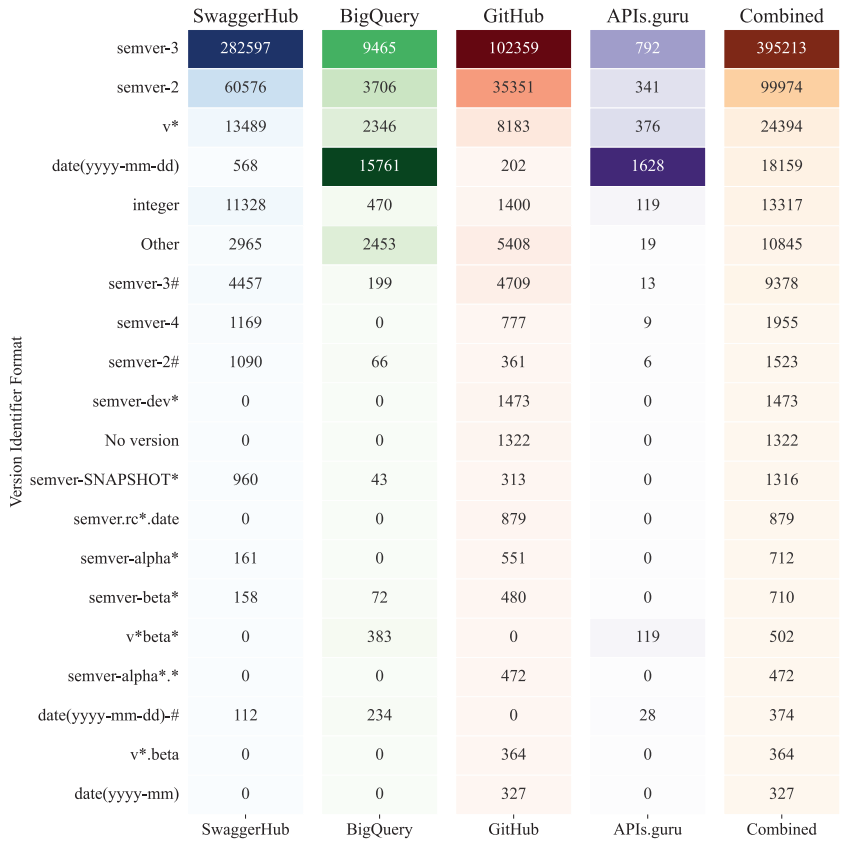

Figure 4 20 most adopted version identifier formats used in metadata **in each of the study datasets and combined**.

Versioning or date-based versioning, and instead, they follow unique, custom formats devised by the API developers. Its presence highlights the diversity and complexity of versioning practices in real-world APIs. It underscores the fact that despite the existence of widely accepted versioning schemes, a considerable number of APIs opt for custom, non-standard versioning formats. However, the use of such formats can lead to inconsistencies, make version management more complex, and potentially hinder the understanding and usage of the API for developers. Therefore, while these non classifiable formats represent a small proportion of the total, it is an important aspect of the versioning landscape that warrants further investigation.

The formats are categorized based on the versioning scheme they adhere to, such as Semantic Versioning (SemVer), date-based versioning, and others. For each format, Table 5 lists the number of occurrences in each dataset.

Table 3 and 4, provide a detailed breakdown of the version identifier formats used across the four datasets.

4.2 URL-based Versioning

URL-based versioning is a method employed by web APIs where the API version is incorporated directly into the URL structure. This version information can be embedded either in the API paths or within the DNS names. When versioning is integrated into paths, it's typically appended as a segment in the URL (e.g., `api.example.com/v1/resource`). Alternatively, with DNS-based versioning, the version is placed in the subdomain or domain (e.g., `v1.api.example.com`).

The deployment implications of these approaches vary. Path-based versioning grants greater flexibility and straightforward resource grouping. However, it can potentially lead to longer URLs as versions accumulate. DNS-based versioning, on the other hand, offers cleaner URLs and enables physical separation of versioned APIs, but requires more elaborate DNS configurations and management.

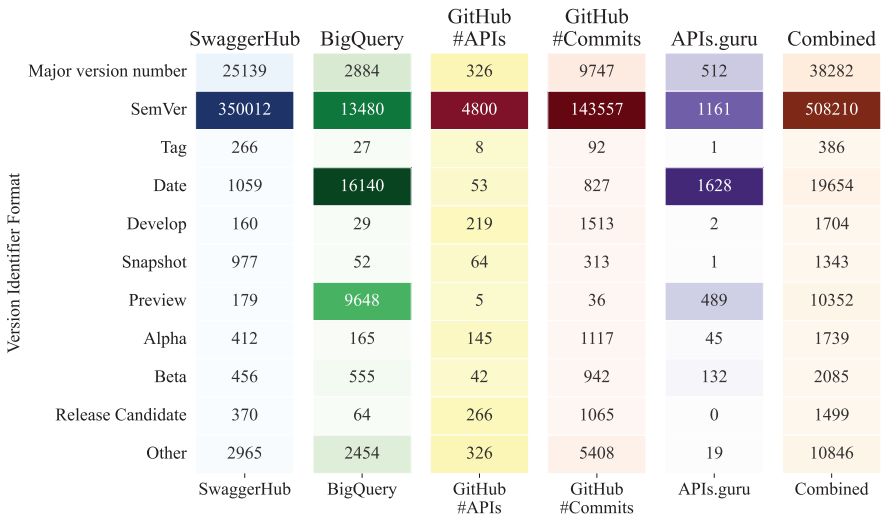

Figure 5 Number of artifacts with version identifiers used in metadata of stable and preview releases **in each of the study datasets and combined**.

4.3 Path-based Versioning

4.3.1 Path-based versioning adoption overview

Table 5 provides an overview of the adoption of path-based versioning across the four datasets, showing that the most commonly used version identifier in path-based versioning is ‘v1‘. The table differentiates between APIs that include version identifiers within each individual path and those that employ a global identifier attached to the server URL. The latter is located within the `servers` field.

Table 3 Number of artifacts with version identifiers used in metadata of stable releases **in each of the study datasets and all combined**

Format	SwaggerHub	BigQuery	GitHub		APIs.guru	Combined
			#APIs	#Commits		
Major version number	25 139	2 884	326	9 747	512	36 608
integer	11 328	470	85	1 400	119	13 402
v*	13 489	2 346	240	8 183	376	25 634
v*#	120	49	5	132	11	317
v*-#	202	19	1	32	6	260
SemVer	350 012	13 480	4 800	143 557	1 161	504 010
semver-2	60 576	3 706	1 146	35 351	341	101 120
semver-2#	1 090	66	19	361	6	1 542
semver-3	282 597	9 465	3 787	102 359	792	397 000
semver-3#	4 457	199	475	4 709	13	9 853
semver-4	1 169	39	13	777	9	2 007
semver-6#	2	3	-	-	-	5
semver-4#	49	-	-	-	-	49
semver-5	60	-	-	-	-	60
semver-5#	6	-	-	-	-	6
semver-6	6	-	-	-	-	6
Tag	266	27	8	92	1	394
latest*	124	13	8	92	1	238
test*	128	12	-	-	-	140
new*	11	-	-	-	-	11
Date	1 059	16 140	53	827	1 628	18 707
date(yyyy-mm)	22	4	19	327	-	372
date(yyyy-mm-dd)	568	15 761	16	202	1 587	18 134
date(yyyy-mm-dd)-#	112	234	3	18	28	395
date(yyyy-mm-ddThh:mm:ssZ)	-	60	16	262	3	341
date(yyyy.mm.dd)	124	46	1	8	7	186
date(yyyymmdd)	112	21	2	10	1	146
date(yyyy.mm)	10	3	-	-	1	14
vdate(yyyy-mm-dd)	8	5	-	-	1	14
date(yyyy)	87	-	-	-	-	87
date(yyyy-mm-dd).hh.mm.ss	1	-	-	-	-	1
date(yyyy-mm-dd hh:mm:ss)	5	1	-	-	-	6

Table 4 Number of artifacts with version identifiers used in metadata of preview releases **in each of the study datasets and all combined**

Format	SwaggerHub	BigQuery	GitHub		APIs.guru	Combined
			#APIs	#Commits		
Develop	160	29	219	1 513	2	1 704
dev*	55	1	2	39	-	95
develop*	10	-	1	1	-	12
semver-dev*	66	12	217	1 473	1	1 552
v*dev*	-	5	-	-	1	6
semver-dev*.*	5	-	-	-	-	5
Snapshot	977	52	64	313	1	1 343
semver-SNAPSHOT*	960	43	64	313	1	1 317
semver-SNAPSHOT*.*	2	-	-	-	-	2
Preview	179	9 648	5	36	489	10 352
date(yyyy-mm-dd)-preview#	77	9 374	1	10	480	9 941
semver-preview*	23	128	1	12	6	169
semver-preview*.*	10	56	3	14	1	81
date(yyyy-mm-dd)-preview*	1	40	-	-	-	41
preview*	10	12	-	-	1	23
semver-pre*.*	1	18	-	-	1	20
semver-pre*	3	-	-	-	-	3
Alpha	412	165	145	1 117	45	1 739
alpha*	50	5	2	27	2	84
semver-alpha*	161	18	25	551	-	730
semver-alpha*.*	45	1	118	472	-	518
v*alpha*	27	96	2	67	42	232
v*p*alpha*	-	2	-	-	1	3
Beta	456	555	42	942	132	2 085
beta*	91	3	2	68	2	164
semver-beta*	158	72	25	480	-	710
semver-beta*.*	27	3	3	10	-	40
v*.beta	3	-	11	364	-	367
v*beta*	24	383	2	20	119	546
semver (beta)	-	7	-	-	-	7
v*p*beta*	-	36	-	-	11	47
Release Candidate	370	64	266	1 065	0	1 499
semver-rc*	176	5	9	107	-	284
rc*	19	1	1	15	-	35
v*rc*	15	2	1	10	-	27
semver-rc*.*	160	56	255	933	-	1 353

• In the SwaggerHub dataset, 15.11% of APIs use path-based versioning with version identifiers in individual paths, and this percentage increases to 19.16% when considering APIs that also use a global identifier in the server URL.

Table 5 Number of artifacts featuring Path-based versioning across datasets

Dataset	Location	#APIs	#Unique	Most Used	#APIs
SwaggerHub	Only paths	57 518 (15,11 %)	464	v1	27 390
	Paths + Servers	72 951 (19,16 %)	957	v1	30 193
BigQuery	Only paths	6 001 (13,20 %)	203	v1	1 947
	Paths + Servers	7 599 (16,71 %)	261	v1	2 605
GitHub	Only paths	1 428 (27,37%)	124	v1	539
		39 860 commits (23,93 %)			16 519 commits
	Paths + Servers	1 793 (34,36%)	180	v1	861
		51 113 commits (30,80 %)			24 055 commits
APIs.guru	Only paths	935 (23,49 %)	138	v1	373
	Paths + Servers	1 381 (34,69 %)	186	v1	480

• The BigQuery dataset shows a similar trend, with 13.20% of APIs using path-based versioning in individual paths, and 16.71% when including APIs with a global identifier.

• The GitHub dataset shows a higher adoption rate of path-based versioning, with 27.37% of APIs using version identifiers in individual paths at some point in their history, and 34.36% when considering APIs with a global identifier.

• The APIs.guru dataset also shows a substantial adoption of path-based versioning, with 23.49% of APIs using version identifiers in individual paths, and 34.69% when including APIs with a global identifier.

4.3.2 Path-based versioning identifiers formats

The results in Table 6 provide an overview of the most adopted version identifier formats appearing in paths across the study datasets. The table presents the formats along with their occurrence in each dataset, expressed as a percentage of the total number of APIs in the respective dataset.

The most common format across all datasets is `v*`, which represents a version number prefixed with the letter 'v'. This format is prevalent in all datasets, with the highest adoption in the GitHub dataset (11.96%), followed by SwaggerHub (8.84%), APIs.guru (10.07%), and BigQuery (5.32%).

The "integer" format, denoting a numeric version identifier, is commonly utilized, particularly in the SwaggerHub dataset. Additionally, the "test*" format, likely indicative of versions used for testing, is prevalent in both the SwaggerHub and BigQuery datasets. Semantic versioning is a favored approach across all datasets, with "semver-2" and "semver-2#" formats frequently used. The "latest*" format, suggesting the most recent API version,

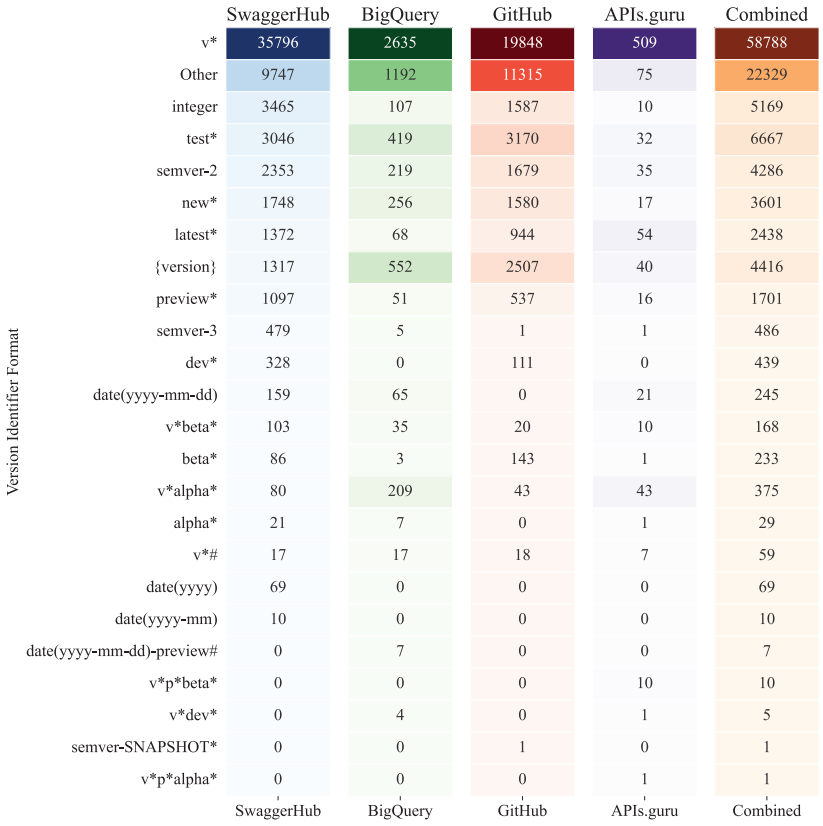

Figure 6 Most frequently adopted version identifier formats appearing in Path **in each of the study datasets and all combined**.

appears less frequently in all datasets. The "version" format, acting as a placeholder for version identifiers, is employed across all datasets but with lower frequency. Notably, the "preview*" format, signifying non-finalized versions, is present in all datasets except BigQuery, while the "alpha*" and "beta*" formats, representing early version stages, see lesser usage across all datasets.

We systematically classified the identified versioning formats into distinct categories, distinguishing between stable and unstable release classes. The heatmap in Table 6 provides a detailed breakdown of the version identifiers used in stable and preview releases across the study datasets. The table categorizes the identifiers into different formats. The table presents the number of artifacts with each format in each dataset.

Table 6 Number of artifacts with Path-based versioning of stable and preview releases Usage of one or multiple format categories in path-based versioning of APIs with multiple versions in production **in each dataset and all combined**

Version Identifier Format	SwaggerHub	BigQuery	GitHub #APIs	GitHub #Commits	APIs.guru	Combined
Major version number	38756	3451	704	20343	692	63242
SemVer	2833	225	36	1680	36	4774
Tag	5932	735	280	5623	95	12385
Date	238	73	0	0	21	332
Develop	338	5	7	111	1	455
Snapshot	0	0	1	1	0	1
Preview	1098	58	23	537	16	1709
Alpha	100	217	3	43	45	405
Beta	190	666	9	163	128	1147
Release Candidate	0	0	0	0	0	0
Other	11064	1744	545	13822	40	26670

From Table 6, it is evident that "Major version number" and "Tag" are the most commonly used formats in stable releases across all datasets (Table 7). This suggests a preference for these formats in stable releases, possibly due to their simplicity and straightforwardness.

In preview releases (Table 8), the "Preview", "Alpha", and "Beta" formats are more prevalent. This indicates that these formats are commonly used to denote the early stages of the version lifecycle, where the API is still under development and not yet finalized.

Regarding the "Others" category in Table 6 and Figure 7, it is worth noting that this category includes version identifiers that do not fit into any of the predefined formats. The high number of artifacts with the "Others" format suggests a diverse range of versioning practices across the study datasets or the fact that the paths are long enough to make it more probable to detect identifiers that are not meant to be used for versioning purposes. This applies also to the case of the "Tag" format.

Further analysis is required to understand the specific characteristics and patterns within the "Others" class. This could involve examining individual API documentation or conducting interviews with API developers to gain insights into their usage purpose.

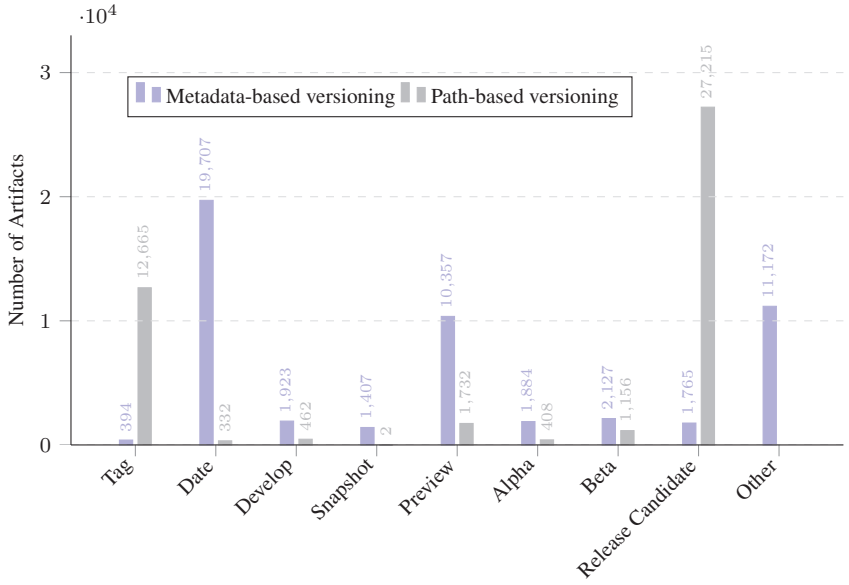

Figure 7 Comparing the adoption of the least used formats classed in metadata-based and path-based versioning **in all datasets combined**. (Semantic Versioning and Major Version Number have been omitted).

4.4 DNS-based Versioning

In DNS-based versioning, the version information is included in the server DNS name.

Table 9 presents the number of APIs that use DNS-based versioning across the four datasets, showing that DNS-based versioning is not as widely adopted as path-based versioning.

Our analysis revealed that not many of the APIs documentation follow the standard URL format RFC 3986. In OpenAPI formats without DNS name, such as ‘‘`/v1/users`’’ or ‘‘`/`’’ are valid values for server URLs.

The low adoption rate of DNS-based versioning showed in Table 9 can be attributed to the fact that not all developers use a URL with the DNS name in the server field.

4.5 Header-based Versioning

In Table 10 and Figure 8, we analyzed the approach’s prevalence across our four datasets, investigated the header names are used to denote API versions, and identified the most common ones.

Table 7 Number of artifacts with Path-based versioning of stable releases in each of the study datasets

Format	SwaggerHub	BigQuery	GitHub		APIs.guru	Combined
			#APIs	#Commits		
Major version number	38 756	3 451	704	20 343	692	63 946
integer	3 465	107	48	1 587	10	5 217
v*	35 796	2 635	668	19 848	509	59 456
v*#	17	17	4	18	7	63
SemVer	2 833	225	36	1 680	36	4 810
semver-2	2 353	219	36	1 679	35	4 322
semver-3	479	5	1	1	1	487
semver-2#	4	1	0	0	0	5
semver-4	1	0	0	0	0	1
Tag	5 932	735	280	5 623	95	12 665
latest*	1 372	256	58	944	54	2 684
new*	1 748	68	88	1 580	17	3 501
test*	3 046	419	141	3 170	32	6 808
Date	238	73	0	0	21	332
date(yyyy-mm-dd)	159	65	0	0	21	245
date(yyyy)	69	0	0	0	0	69
date(yyyy-mm)	10	0	0	0	0	10
date(yyyymmdd)	1	0	0	0	0	1
date(yyyy-mm-dd)-#	0	1	0	0	0	1

The results of our analysis indicate that the header-based versioning approach is not as prevalent as path-based versioning. A wide variety of header names used in the SwaggerHub dataset (Figure 8), indicating a lack of standardization among the APIs adopting that practice.

In Figure 8 we depict the adoption of header-based versioning in the SwaggerHub dataset over the years.

4.6 Dynamic Versioning

In this paper we examine the prevalence of APIs that offer endpoints for retrieving either the current version or the list of available versions of the API. Our work also studies the potential correlations between the adoption of dynamic versioning strategies and the utilization of header-based versioning in real-world APIs. In Table 11 we show the number of APIs having one endpoint dedicated to fetch the current version/versions of the API.

The `GET /version` endpoint, which retrieves the current version of the API, is more prevalent across all datasets. In the SwaggerHub dataset, 1185 APIs provide this endpoint, while in the BigQuery, GitHub, and APIsguru

Table 8 Number of artifacts with Path-based versioning of preview releases **in each of the study datasets and all combined**

Format	SwaggerHub	BigQuery	GitHub		APIs.guru	Combined
			#APIs	#Commits		
Develop	338	5	7	111	1	462
dev*	328	1	7	111	-	447
develop*	10	-	-	-	-	10
v*dev*	-	4	-	-	1	5
Snapshot	0	0	1	1	0	2
semver-SNAPSHOT*	-	-	1	1	-	2
Preview	1 098	58	23	537	16	1 732
preview*	1 097	51	23	537	16	1 724
date(yyyy-mm-dd)-preview#	-	7	-	-	-	7
semver-preview*.*	1	-	-	-	-	1
Alpha	100	217	3	43	45	408
v*alpha*	80	209	3	43	43	378
alpha*	21	7	-	-	1	29
v*p*alpha*	-	1	-	-	1	2
Beta	190	666	9	163	128	1 156
beta*	86	3	5	143	1	238
v*beta*	103	628	4	20	117	872
v*p*beta*	-	35	-	-	10	45
v*.beta	1	-	-	-	-	1
Release Candidate	0	0	0	0	0	0

Table 9 Number of APIs with DNS-based versioning

Dataset	#APIs	#Distinct Version IDs	Most used ID	Occurrence (#APIs)
GitHub	3 (215 commits)	1	{`version`}	3
BigQuery	21	2	{`version`}	19
SwaggerHub	64	8	{`version`}	57
APIs.guru	10	1	{`version`}	10

datasets, 435, 67, and 11 APIs provide this endpoint, respectively. On the other hand, the `GET /versions` endpoint, which retrieves the list of available versions of the API, is less common. In the SwaggerHub dataset, only 153 APIs provide this endpoint. Similarly, in the BigQuery, GitHub, and APIsguru datasets, only 17, 4, and 8 APIs provide this endpoint, respectively. These results suggest that while some APIs provide dynamic versioning capabilities, the majority of APIs prefer to provide only the current version information. This could be due to the simplicity and lower maintenance

Table 10 Adoption of header-based versioning across the study datasets: BigQuery, GitHub and APIs.guru

BigQuery	#APIs
x-ms-version	6
x-api-version	2
x-ph-api-version	1
x-amz-fwd-header-x-amz-version-id	1
accept-version	1
GitHub	#APIs
basiq-version	3
cdi-version	1
x-myobapi-version	1
apiversion	1
version	1
x-api-version	1
APIs.guru	#APIs
x-ms-version	3
x-readme-version	1
trakt-api-version	1
x-amz-fwd-header-x-amz-version-id	1
x-je-api-version	1
zuora-version	1

Header Name	#APIs
api-version	84
x-api-version	73
app-version	22
x-app-version	17
Accept-version	13
appversion	9
accept-version	8
x-version	8
app_version	6
x-version-api	5
126 Distinct Names	549

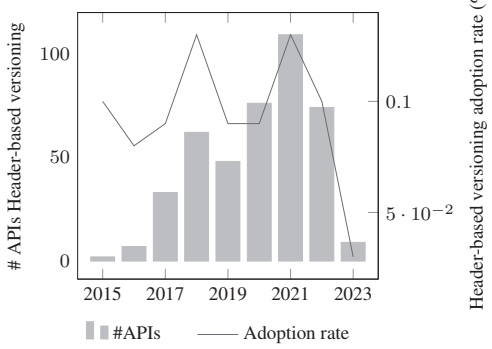

Figure 8 Header-based adoption in SwaggerHub Dataset over the years.

overhead of only managing a single current version. However, providing a list of available versions can offer more flexibility to the clients, allowing them to choose the most suitable version for their needs.

Table 11 Number of artifacts where dynamic version information endpoints is detected

Endpoint	SwaggerHub	BigQuery	GitHub		APIsguru
			#API	#Commits	
GET /version	1185	435	67	2585	11
GET /versions	153	17	4	438	8

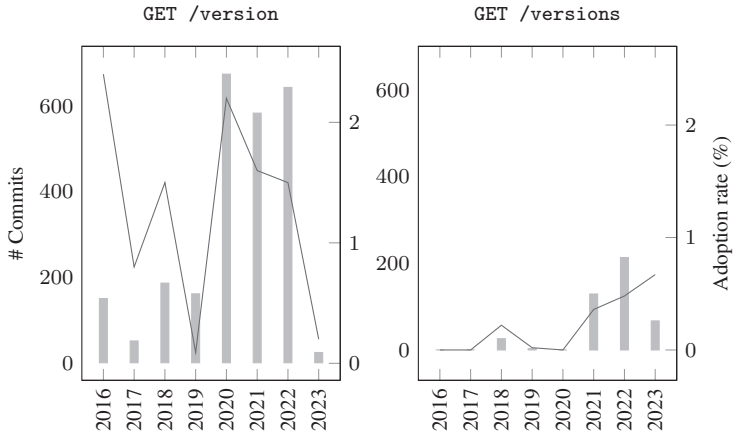

Figure 9 Dynamic versioning over the years in Github Dataset.

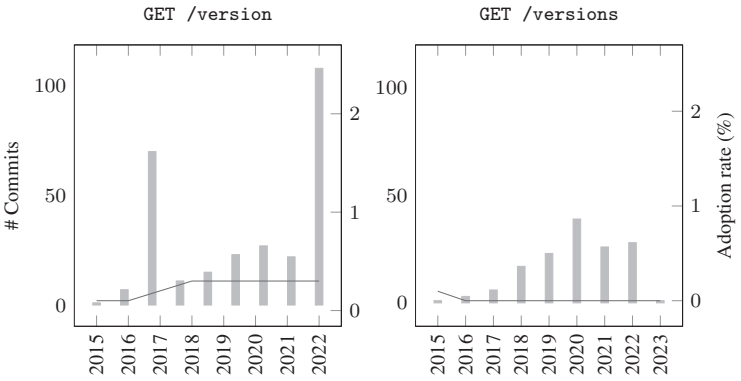

Figure 10 Dynamic versioning over the years in SwaggerHub Dataset.

Figures 9 and 10 illustrate the adoption of dynamic versioning over the years in the GitHub and SwaggerHub datasets, respectively. Figure 9, the adoption of the `GET /version` endpoint in the GitHub dataset has been relatively very low and non-stable over the years. On the other hand, the

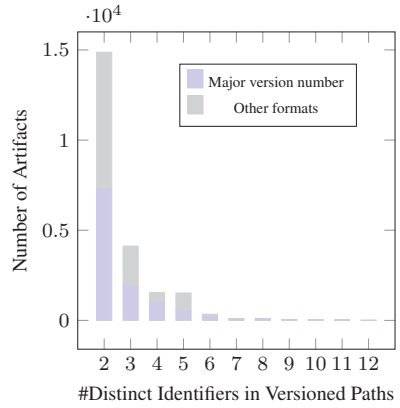

Figure 11 The adoption of major version number vs other formats in identifiers found in the paths of APIs with multiple versions in production **in all datasets combined.**

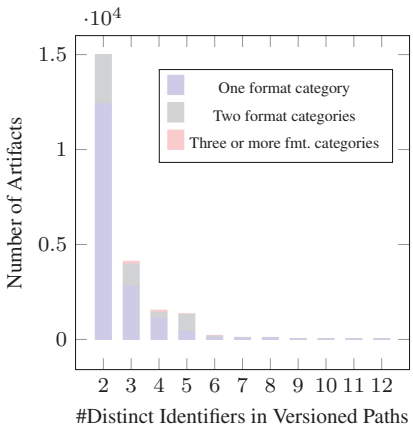

Figure 12 Usage of one or multiple format categories in path-based versioning of APIs with multiple versions in production **in all datasets combined**.

adoption of the `GET /versions` endpoint has been minimal, with a small increase starting from 2020. In Figure 10, we can observe that the adoption of the `GET /version` endpoint in the GitHub dataset has been relatively very low and stable over the years. On the other hand, the adoption of the `GET /versions` endpoint has been minimal all the time.

It is also worth noting that the adoption of dynamic versioning strategies does not seem to correlate with the use of header-based versioning. We anticipated discovering a correlation between the utilization of header-based

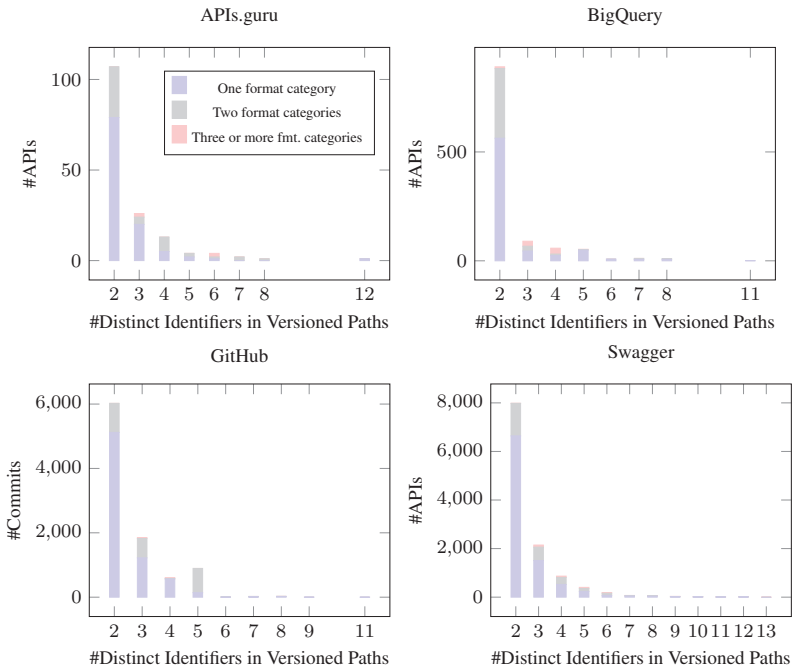

Figure 13 Usage of one or multiple format categories in path-based versioning of APIs with multiple versions in production.

versioning and dynamic versioning. However, our analysis revealed only seven APIs (comprising six from BigQuery and one from SwaggerHub) where these two practices were employed concurrently.

We looked at the joint usage of dynamic versioning and query parameters where the clients can send the version identifier. Within the APIs that feature dynamic versioning, we found 146 APIs in SwaggerHub, 320 APIs in BigQuery, 12 APIs in GitHub dataset (223 commits), and three APIs in APIs.guru dataset that have version query parameters in at least one operation.

4.7 "Two in Production" Evolution Pattern

We analyzed the usage of the "two in production" evolution pattern [21, 33] across the four study datasets by examining the APIs that have paths with distinct version identifiers.

The examination involved an analysis of specifications containing descriptions of various API versions. This analysis was predicated on the assumption that the presence of multiple versions within these specifications implied the coexistence of these API versions in a production environment.

As demonstrated in the bar charts of Figures 11 and 12, our analysis revealed the presence of 22 632 adoptions the "two in production" evolution pattern: 11 870 in SwaggerHub, 9 465 commits in GitHub, 1 139 in BigQuery, and 158 in APIs.guru collection (See Figures 13 and 14). Notably, among these APIs, 419 APIs from BigQuery and 219 APIs from SwaggerHub exhibited the noteworthy characteristic of using different formats for each version.

As illustrated in Figure 11, for APIs maintaining two versions in production, approximately 51% employ the Major Version Number as the format for the version identifier within the paths. This rate of adoption remains consistent for scenarios involving three to six concurrent versions. Beyond this range, the Major Version Number becomes the sole versioning format utilized.

The results presented in Figure 13 provide insights into the usage of multiple format categories in path-based versioning of APIs that adopt the "two in production" evolution pattern in each dataset sparately, where we can see that the adoption of Major Version Number slightly differs in the case of BigQuery.

In the APIs.guru dataset, the majority of APIs (79) use only one format category, while a smaller number (28) use two format categories. Only a very small number of APIs (2) use three or more format categories.

A similar pattern is observed in the BigQuery dataset, with a majority of APIs (563) using one format category, a smaller number (320) using two format categories, and a very small number (8) using three or more format categories.

In the GitHub dataset, the majority of APIs (5 120) use one format category, while a smaller number (902) use two format categories. Only a very small number of APIs (31) use three or more format categories.

In the Swagger dataset, the majority of APIs (6641) use one format category, while a smaller number (1339) use two format categories. A slightly larger number of APIs (87) in this dataset use three or more format category compared to the other datasets.

While the use of multiple format categories in path-based versioning is not uncommon, the majority of APIs prefer to use a single format category.

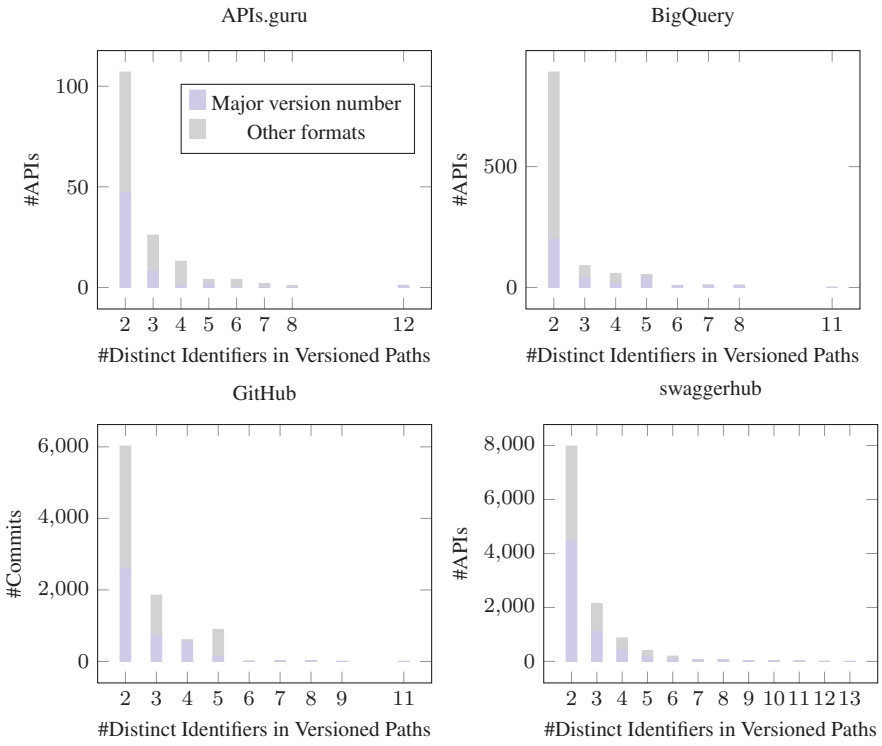

Figure 14 Comparing the adoption of major version number vs other formats in identifiers found in the paths of APIs with multiple versions in production.

This could be due to the simplicity and consistency offered by using a single format category.

In Figure 12, we quantify the number of APIs employing precisely one, two, or three or more format combinations for APIs with more than one version in production. It is evident that, in the majority of instances, APIs tend to use no more than one format for versioning.

The results presented in Figure 14 provide a comprehensive overview of the version formats used in APIs that adopt the "two in production" evolution pattern in each of the datasets separately.

In the APIs.guru dataset, the majority of APIs (4470) use the "Major version number" format, while a smaller number (3511) use other formats. This trend is also observed in the BigQuery dataset, with a majority of APIs (2611) using the "Major version number" format, and a smaller number (3411) using other formats.

In the GitHub dataset, a similar pattern is observed, with a majority of APIs (199) using the "Major version number" format, and a smaller number (692) using other formats. However, in the Swagger dataset, the use of the "Major version number" format (4470) is almost equal to the use of other formats (3511).

These results suggest that while the `Major version number` format is the most commonly used format in path-based versioning, a significant number of APIs also use other formats. This could be due to the flexibility and adaptability offered by these other formats, allowing API developers to tailor their versioning strategy to the specific needs and requirements of their API.

4.8 Version Formats Adoption Over the Years

Figure 15 shows that in 2015 Semantic Versioning (SemVer) held sway as the predominant versioning format, constituting the choice in 59% of the analyzed APIs, while the utilization of the Major Version Number format accounted for 38.11% of the cases. However, an observable shift occurred in the subsequent years. Notably, there was a conspicuous decline in the adoption of the simplified format, characterized by solely the major version number, accompanied by a notable resurgence in SemVer adoption during the year 2017. Nonetheless, the substantial surge in SemVer adoption observed in 2017 was not sustained in the subsequent years. Instead, SemVer's adoption exhibited a relatively stable trajectory over the years, punctuated by occasional slight declines noted in 2021 and 2022.

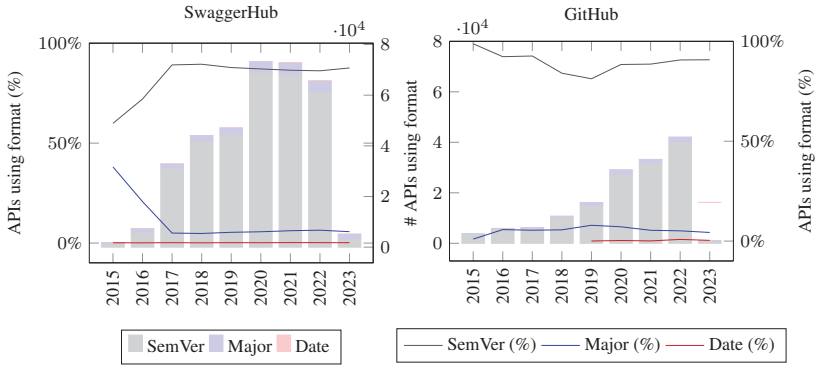

Figure 15 Adoption of Semantic Versioning, Major version number and Date formats over the years.

5 Results Summary

Q1: *What are the commonly adopted practices for Web APIs versioning?* Based on our analysis investigating the adoption of the versioning practices: metadata-based, URL-based, header-based, and dynamic versioning, we detected the usage of these practices with different frequencies across the four study datasets. Metadata-based versioning, where the version information is included in the API metadata, is prevalent in 98% of the APIs. This practice is favored due to its simplicity and the ease of managing version information in a centralized location. URL-based versioning, where the version information is included in the URL of the API, is adopted in 26.23% of the APIs. This practice offers the advantage of making the version information immediately visible and accessible to the clients. Header-based versioning, where the version information is included in the HTTP headers, is less common, being used in only 0.17% of the APIs. This could be due to the additional complexity it introduces in managing version information. Lastly, dynamic versioning, where the version information is discovered dynamically at runtime, is used in a minority of APIs (1088 APIs in all datasets).

Q2: *How do developers distinguish stable from preview releases?* Developers distinguish between stable and preview releases primarily through the use of specific version identifier formats. Our analysis of the study datasets revealed that the"Major version number" and "Tag" formats are the most commonly used in stable releases across all datasets. This suggests a preference for these formats in stable releases, possibly due to their simplicity and straightforwardness. In contrast, the "Preview", "Alpha", and "Beta" formats are more prevalent in preview releases. This indicates that these formats are commonly used to denote early stages of the version lifecycle, where the API is still under development and not yet finalized. For instance, in the SwaggerHub dataset, the "semver-beta*.*" format was used in 27 APIs, the "v*.beta" format in 3 APIs, and the "v*beta*" format in 24 APIs.

Q3: *To what extent is the practice of semantic versioning adopted in Web APIs, and are there alternative versioning schemes in use?* Semantic versioning (SemVer) is found to be a widely adopted practice in Web APIs. Our analysis shows that in 2015, SemVer was the predominant versioning format, used in 59% of the analyzed APIs. However, its adoption has seen some fluctuations over the years. For instance, in 2017, there was a significant increase in SemVer adoption, reaching 89.12% of the APIs. However, this surge was not sustained in the subsequent years, with a slight

decline noted in 2021 and 2022. Despite these fluctuations, SemVer remains a popular choice, with its adoption rate in 2023 standing at 87.60%.
In terms of alternative versioning schemes, the Major Version Number format is the second most common, used in 38.11% of APIs in 2015. However, its adoption has seen a decline over the years, dropping to 5.70% in 2023. Another alternative is the Date format, which, although less common, has seen a slight increase in adoption, from 0.17% in 2015 to 0.36% in 2023.
These findings suggest that while SemVer is the most prevalent versioning scheme, there is a diversity of practices in Web API versioning, with some APIs opting for alternative schemes such as the Major Version Number or Date formats.

Q4: *What is the prevalence of APIs with multiple versions in production? how many concurrent versions exist?*
Our analysis shows the presence APIs have multiple versions in production concurrently in all the datasets. Specifically, 14.29% of the APIs in the SwaggerHub dataset, 5.50% in the BigQuery dataset, 6.99% in the GitHub dataset, and 3.96% in the APIs.guru dataset have multiple versions in production.
In terms of the number of concurrent versions, our analysis reveals a wide range. The majority of APIs with multiple versions in production have between 2 to 5 concurrent versions. However, there are also APIs with a high number of concurrent versions. For instance, in the SwaggerHub dataset, the maximum number of concurrent versions found in an API is 13. This suggests that some APIs maintain a large number of versions in production, possibly to cater to a wide range of clients with different version requirements.

Q5: *How has the adoption of dynamic versioning and header-based versioning practices evolved over time?*
Our analysis reveals interesting trends in the adoption of dynamic versioning and header-based versioning practices over time. Dynamic versioning, despite its potential benefits of flexibility and adaptability, is used in a minority of APIs across all datasets. This could be attributed to the additional complexity and overhead associated with managing dynamic version information.
On the other hand, header-based versioning, where the version information is included in the HTTP headers, is even less common. This could be due to the additional complexity it introduces in managing version information. However, it is worth noting that the adoption of dynamic versioning strategies does not seem to correlate with the use of header-based versioning. We anticipated discovering a correlation between the utilization of header-based

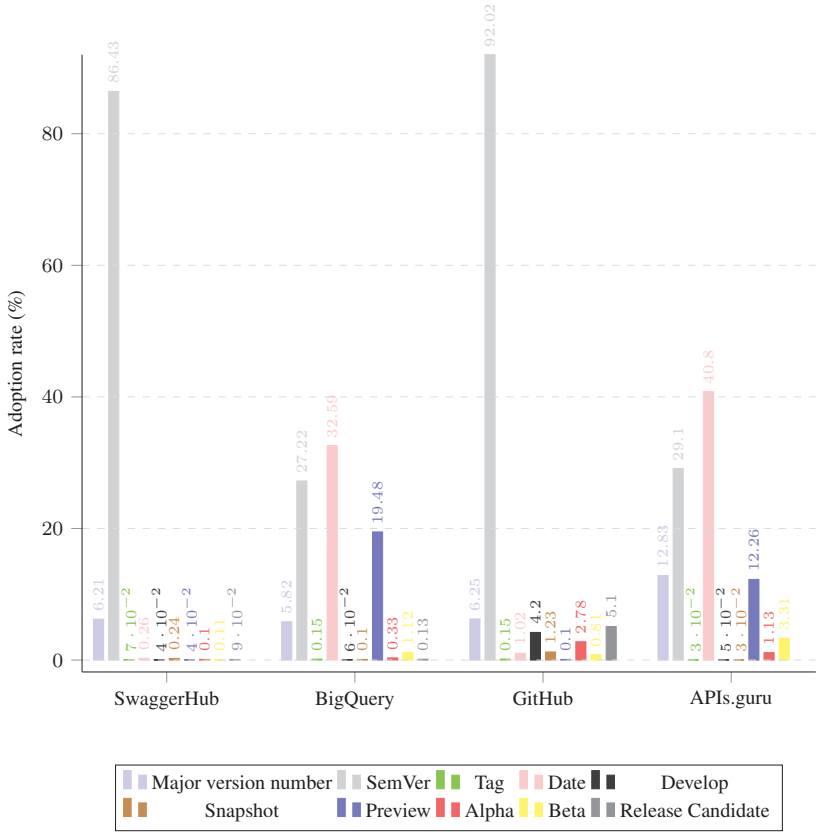

Figure 16 Adoption of different format categories in each dataset in Meta-data based versioning.

versioning and dynamic versioning. However, our analysis revealed that this correlation was non existent.

While there is a diversity of practices in Web API versioning, the adoption of dynamic and header-based versioning practices has remained relatively low over the years. This highlights the need for further research to understand the factors influencing these adoption trends.

Q6: *How sensitive are the results to the source of the API descriptions?* The results show some sensitivity to the source of the API descriptions. For instance, the adoption rate of Semantic Versioning (SemVer) in the SwaggerHub dataset was 87.60% in 2023, while in the GitHub dataset, it was 90.77%. Similarly, the adoption rate of the Major Version Number format

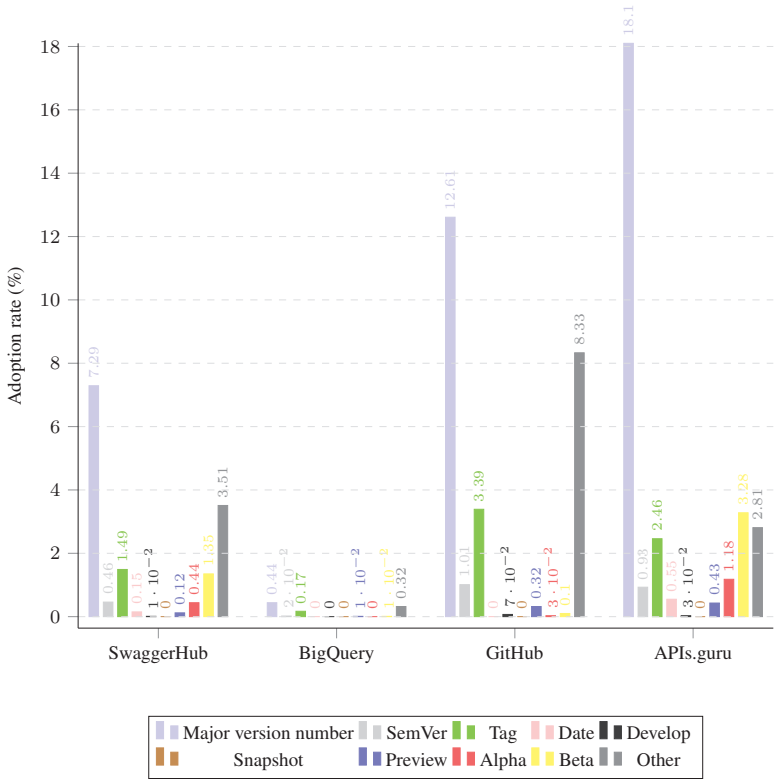

Figure 17 Adoption of different format categories in each dataset in Path-based versioning.

was 5.70% in the SwaggerHub dataset and 4.32% in the GitHub dataset in 2023.

Figure 16, depicts the adoption of identifiers format in Metadata-based versioning in each of the four datasets. The adoption dominance of each format follows the same order in the case of BigQuery and APIs.guru datasets, where the most common one is “Date”, followed by “Semever”. Another particular noticed aspect is the presence of a relatively high number of “Preview” identifiers.

The higher prevalence of the “Preview” identifiers in the BigQuery and APIs.guru datasets could be indicative of the experimental nature of many APIs on these data sources.

On the other hand, the SwaggerHub and GitHub datasets show a higher prevalence of the “SemVer” identifiers.

In terms of the "Major Version Number" format, its adoption is relatively consistent across all of SwaggerHub, GitHub and BigQuery datasets, ranging from 5.82% in the GitHub dataset to 6.25% in GitHub. While in APIs.guru dataset, the adoption rate of this format goes up to 12.83%.
Figure 17 the adoption of different version formats in APIs using path-based versioning accross the four datasets.
These differences suggest that while the overall trends in versioning practices are similar across different sources of API descriptions, there are some variations in the specific adoption rates. This could be due to differences in the communities of developers contributing to these sources, their preferences, and their familiarity with different versioning schemes.

6 Web API versioning in OpenAPI 4.0: proposal

The diversity of formats found in in the `info.version` field of the OpenAPI specification is due to the way to document API versioning in OpenAPI, which does not provide any information about the type of versioning adopted by the API, such as semantic versioning, date-based versioning, or custom versioning. It also does not explicitly support the use of multiple versions of the API specification in the same document, which can be useful for documenting deprecated or experimental features.

Introducing standardized metadata fields for version identifiers in the OpenAPI specification would significantly enhance *clarity* and *interoperability* across web APIs. By clearly defining the type of versioning adopted—be it semantic, date-based, or custom—developers and tools can more easily understand and manage API versions. This standardization would facilitate automated tools in accurately interpreting version changes, thereby improving API documentation, and would aid in the seamless integration of APIs with differing versioning schemes.

In our proposal in Listing 2, the existing `info.version` field would change type from string to an object that comprises the `value` field to specify the version identifier string, the schema field to document and enforce a precise, structured version format, and the upgrade field to define the version upgrade rules that should be followed, depending on the chosen format. In addition, for recording the release date, we introduce a timestamp so that the age of the API release can be tracked explicitly. Likewise, tags can represent the lifecycle phase to which the artifact belongs. A separate build counter can complement the timestamp so that DevOps pipelines can use a fine-grained

identifier to stamp each artifact version without affecting the main version identifier.

Listing 2 Proposal for web API version in OpenAPI

```
version:
    semantic-identifier: 1.2.3  # Semantic version identifier
        ↪ value
    lifecyle: "stable" # stable, preview, rc, alpha, beta
    timestamp: YYYY-MM-DD HH:MM:SS # optionally track the API
        ↪ release age
    build: NNNN  # an integer build counter
```

By knowing the versioning strategy (such as semantic versioning, date-based versioning, or custom versioning), API consumers can better anticipate the nature of changes and updates. This aids in planning for potential compatibility issues and migration efforts. The specification can serve as a reference point for all API development team members, making it clear how version numbers are assigned and what each increment signifies.

7 Related Work

In our previous paper [27], we evaluated the prevalence of Semantic Versioning (SemVer) in API versioning, focusing on the utilization of the `info.version` field in stable API releases committed to GitHub between 2015 and 2022. Our findings indicated a strong adoption rate of SemVer in stable releases, averaging at 75.84% $\pm$ 4.79%. In this paper, we study the adoption of all of SemVer, Major Version Number, and Date formats using he largest dataset we have which is the SwaggerHub. A notable characteristic of this dataset is the inclusion of key temporal markers for each API specification:`created_at`, which could be extracted using the SwaggerHub API. These dates denote the precise moments of the specification's creation date

In previous work on Web API evolution [13] we studied the API size changes over time, without considering how developers tend to summarize these changes through versioning. Other studies have investigated the relationship between software changes and versioning for software libraries published in package management tools [23, 32], our work takes a different approach by focusing exclusively on the evolution of the interface due to the limitations and challenges posed by the lack of access to the corresponding backend implementation code for Web APIs. These results highlight the need for further research on the impact of different versioning practices on API and backend development.

In the study conducted by Dietrich et al. [14], the authors aimed to analyze versioning practices in software dependency declarations. To do this, they leveraged a rich dataset collected from the `libraries.io` repository, which contained metadata from 71 884 555 packages published on 17 different package management platforms, including Home-brew, Maven, and NPM, along with their respective dependency information. The authors employed a similar approach with detectors based on regular expressions to categorize the dependency versions into 13 different formats. Their findings revealed that the majority of package managers predominantly use flexible dependency version syntax, with a considerable uptake of semantic versioning in case of Atom, Cargo, Hex, NPM, and Rubygems. Additionally, a survey of 170 developers showed that they rarely modified the declared dependencies' version syntax as the project evolved.

In a separate study [12], the author focused on the versioning practices adopted by developers when using continuous integration services such as GitHub Actions. The results indicated that 89.9% of the analyzed version tags followed GitHub's recommendation of only referring to the major version in the identifier, with only a small fraction (0.9%) including minor version information and 9.2% using the SemVer-3 format. This differs from our findings, where we found that SemVer-3 was the most widely adopted semantic versioning format.

Several studies investigated the correlation between software changes and versioning in the case of libraries published in packages manager tools [23,26,32]. The most recent one is by L.Ochoa et al. [23] who replicated a previous study performed by Raemaekers et al. [26] studying the level of adherence to semantic versioning and the impact of breaking changes on clients. They found that 83.4% of all packages upgrades do comply with semantic versioning principles. And, 20.1% of non-major releases are actually introducing backward incompatible changes. They also found an increase of adherence and reduction of semantic versioning misuses: from 67.7% non-major breaking releases in 2005 to 16.0% in 2018. Another recent study on the accuracy of semantic version upgrades is [32], where the authors did not only study the changes happening at the API level, but looked at the internal code behaviour to understand the breaking problems happening in the case of compatible version upgrades (when only `MINOR` or `PATCH` are upgraded). By analysing 180 real-world examples with breaking problems, they computed the probabilities of different types of changes (e.g: additional parameter, additional branch, additional try/catch) to trigger backward incompatibility issues.

In [20], the authors studies the compliance to SemVer in the GoLang ecosystem, focusing on breaking changes and their impacts. The authors developed a tool, GoSVI (Go Semantic Versioning Insight), to detect breaking changes by analyzing the types of identifiers in client programs and comparing them withh breaking changes. They collected a dataset from GitHub, including 124K third-party libraries (TPLs) and 532K client programs, to analyze SemVer compliance and the impact of breaking changes. Their findings indicate that 86.3% of library upgrades follow SemVer compliance, yet 28.6% of non-major upgrades introduce breaking changes. They also found an improvement in SemVer compliance over time and identified that 33.3% of downstream client programs could be affected by breaking changes.

8 Conclusion

Versioning in Web APIs is a fundamental practice to ensure their compatibility and ease their maintainability. In this empirical study we focused on version identifiers, observing their representation formats, static or dynamic discoverability, and purpose across 4 different datasets of 602'859 OpenAPI descriptions. The vast majority utilized static versioning in the API metadata (592 033; 98%), while only a subset include version identifiers embedded in the API endpoint path URL addresses (133 456; 22%). Only a small fraction (4 219; 0.7%) supported dynamic discovery of the current version through a dedicated endpoint.

In terms of version format, we identified 10 221 distinct version identifiers and 257 distinct formats used to distinguish stable and preview releases, with 23 251 pre-release versions across different stages of the API release lifecycle. While most APIs use semantic version identifiers to indicate the expected impact on clients of changes with respect the previous version, a few instead use version identifiers to track the age of the API.

We also observed the usage of the "two in production" evolution pattern in 13501 APIs (4 250 with more than 2 versions): 158 in APIs.guru, 1139 in BigQuery, 365 (with 9 465 commits) in GitHub, and 11 839 in SwaggerHub. In these cases, the most prevalent format for version identifiers attached to the path was to reference only the major version.

As future work, we plan to further investigate the adherence of developers to semantic versioning guidelines and study the types of API changes that drive major or minor version changes.

Acknowledgements

This work is funded by the SNSF, with the API-ACE project nr. 184692.

References

[1] Semantic Versioning. https://semver.org/.
[2] https://github.com/USI-INF-Software/API-Versioning-practices-detection.
[3] Vercel API. https://vercel.com/docs/rest-api/endpoints.
[4] OpenAPI Initiative. https://www.openapis.org/.
[5] SwaggerHub API. https://app.swaggerhub.com/apis-docs/swagger-hub/registry-api/1.0.67.
[6] https://docs.npmjs.com/about-semantic-versioning.
[7] Release naming conventions. https://www.drupal.org/node/1015226.
[8] https://docs.fedoraproject.org/en-US/packaging-guidelines/Versioning/.
[9] Google BigQuery. https://github.com/topics/bigquery.
[10] Contextual Content Discovery: You've forgotten about the API endpoints. https://blog.assetnote.io/2021/04/05/contextual-content-discovery/.
[11] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: cost negotiation and community values in three software ecosystems. In *Proc. 24th International Symposium on Foundations of Software Engineering*, pages 109–120, 2016.
[12] Alexandre Decan, Tom Mens, Pooya Rostami Mazrae, and Mehdi Golzadeh. On the use of github actions in software development repositories. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 235–245, 2022.
[13] Fabio Di Lauro, Souhaila Serbout, and Cesare Pautasso. A large-scale empirical assessment of web api size evolution. *Journal of Web Engineering*, 21(6):1937–1980, 2022.
[14] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. Dependency versioning in the wild. In *Proc. 16th International Conference on Mining Software Repositories (MSR)*, pages 349–359, 2019.
[15] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web api growing pains: Loosely coupled yet strongly tied. *Journal of Systems and Software*, 100:27–43, 2015.

[16] Anthony Giretti. API versioning. In *Beginning gRPC with ASP.NET Core 6*, pages 223–237, 2022.

[17] Elias Grünewald, Paul Wille, Frank Pallas, Maria C Borges, and Max-R Ulbricht. Tira: an openapi extension and toolbox for gdpr transparency in restful architectures. In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 312–319. IEEE, 2021.

[18] Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. Web api evolution patterns: A usage-driven approach. *Journal of Systems and Software*, 198:111609, 2023.

[19] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. How does web service api evolution affect clients? In *2013 IEEE 20th International Conference on Web Services*, pages 300–307. IEEE, 2013.

[20] Wenke Li, Feng Wu, Cai Fu, and Fan Zhou. A large-scale empirical study on semantic versioning in golang ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1604–1614. IEEE, 2023.

[21] Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. Interface evolution patterns: balancing compatibility and extensibility across service life cycles. In *Proc. 24th EuroPLoP*, 2019.

[22] Klaus Marquardt. Patterns for software release versioning. In *Proc. of the 15th European Conference on Pattern Languages of Programs (EuroPLoP)*, 2010.

[23] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. Breaking bad? semantic versioning and impact of breaking changes in maven central. *Empirical Software Engineering*, 27(3):1–42, 2022.

[24] Lianyong Qi, Houbing Song, Xuyun Zhang, Gautam Srivastava, Xiaolong Xu, and Shui Yu. Compatibility-aware web api recommendation for mashup creation via textual description mining. *ACM Transactions on Multimidia Computing Communications and Applications*, 17(1s):1–19, 2021.

[25] Mikko Raatikainen, Elina Kettunen, Ari Salonen, Marko Komssi, Tommi Mikkonen, and Timo Lehtonen. State of the practice in application programming interfaces (APIs): A case study. In *Proc. 15th European Conference on Software Architecture (ECSA)*, pages 191–206, 2021.

[26] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software*, 129:140–158, 2017.

[27] Souhaila Serbout and Cesare Pautasso. An empirical study of web api versioning practices. In *International Conference on Web Engineering*, pages 303–318. Springer, 2023.

[28] SM Sohan, Craig Anslow, and Frank Maurer. A case study of web api evolution. In *2015 IEEE World Congress on Services*, pages 245–252. IEEE, 2015.

[29] Aimilios Tzavaras, Nikolaos Mainas, Fotios Bouraimis, and Euripides GM Petrakis. Openapi thing descriptions for the web of things. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1384–1391. IEEE, 2021.

[30] Aimilios Tzavaras, Nikolaos Mainas, and Euripides GM Petrakis. Openapi framework for the web of things. *Internet of Things*, 21:100675, 2023.

[31] Jinqiu Yang, Erik Wittern, Annie TT Ying, Julian Dolby, and Lin Tan. Towards extracting web api specifications from documentation. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, pages 454–464, 2018.

[32] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. Has my release disobeyed semantic versioning? static detection based on semantic differencing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[33] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. *Patterns for API Design - Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley, 2022.

Biographies

Cesare Pautasso is full professor at the Software Institute at USI Lugano, Switzerland. He leads the Architecture, Design and Web Information

Systems Engineering research group, which focuses on building experimental systems to explore the intersection of Text-to-Visual modeling languages, API analytics and liquid software architectures. He was the program co-chair for EuroPLoP 2023, ICWE 2021, ICSOC 2013, ECOWS 2010 and Software Composition 2008 and the EuroPLoP 2022, ICWE 2016, ECOWS 2011 general chair. He is the co-editor of the IEEE Software Insights department. His e-books on Email Anti-Patterns, Software Architecture, Business Process Management, and API visualization are available on LeanPub.

Souhaila Serbout is a Ph.D. candidate at the Software Institute (USI) in Lugano, Switzerland, under the supervision of Prof. Dr. Cesare Pautasso. She carried out a Masters's in New Technologies of Informatics at the Faculty of Informatics of the University of Murcia, Spain. In 2018, obtained a state engineer's degree in computer science from Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes in Rabat, Morocco. Currently, she works on visualizing and analyzing Web APIs structures and data models to find the mismatches between developers' expectations and real-world APIs designs.