



An empirical study of Web API versioning practices

Souhaila Serbout  and Cesare Pautasso 

Software Institute (USI), Lugano, Switzerland
`first-name.last-name@usi.ch`

Abstract. As Web APIs evolve, developers assign them version identifiers to reflect the amount and the nature of changes that the API clients should expect. In this work we focus on identifying versioning practices adopted by Web API developers by extracting and classifying version identifiers found in a large collection of OpenAPI descriptions. In particular, we observe how frequently different versioning schemes have been adopted for identifying both stable and preview releases (e.g., simple version counters, semantic versioning, or release timestamps). We further study the stability of versioning schemes during APIs evolution. We also detect APIs which offer dynamic access to versioning metadata through dedicated endpoints as well as APIs which support clients expecting to reach up to 14 different versions of the same API at the same time. Overall the results offer a detailed view over current Web API versioning practices and can serve as the basis for future discussions on how to standardize critical API versioning metadata.

Keywords: API · Web API · OpenAPI · Empirical Study · Versioning

1 Introduction

The evolution of Web APIs requires versioning practices to ease compatibility checking and maintainability for both API providers and clients [13,19]. 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 [14].

The lack of a centralized registry for Web APIs, combined with the flexibility for service providers to use their own versioning approaches [17], has led to multiple and sometimes inconsistent practices in terms of discoverability and notification of breaking changes [9]. Such variability in versioning practices raises questions about the prevalence of semantic versioning [1] adoption among Web APIs. In this study, we aim to classify the versioning schemes used for Web APIs and to track how their adoption of changes over time and across the API release cycle. Additionally, we aim to examine the frequency and extent of concurrent availability of multiple API versions, as the introduction of backward incompatible changes in web APIs can have negative impacts on clients, unless both

old and new versions are kept in production [14]. To achieve this, we analyze a large dataset of 186,259 OpenAPI descriptions mined from GitHub, tracing the change histories of 7114 APIs, to answer the following research questions:

Q1: What is the prevalence of versioning in Web APIs? How often is version information located outside of the API metadata or discovered dynamically?

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: How often do developers switch to different versioning schemes during the lifespan of their APIs?

Q5: Has the adoption of semantic versioning changed over the past few years?

Q6: What is the prevalence of APIs with multiple versions in production? how many concurrent versions exist, and which formats are used in this case?

Answering these questions will provide valuable observations on the state of the practice regarding Web API versioning. Given the simple approach to versioning in the OpenAPI specification, there is room for different interpretations on how to encode whether an API is a stable or preview release, and different version identification strategies are possible. Our study results reveal a need for more detailed versioning metadata in the OpenAPI standard [2]. Likewise, we did not only observe the presence of a variety of formats to represent static version identifiers, but also emerging support for dynamic version discovery, as well as two or even more (up to 14) coexistent versions in production.

The remainder of this paper is organized as follows. In Section 2 we define basic versioning concepts and how they are expressed in OpenAPI. In Section 3, we describe the methodology used to collect the API artifacts. In Section 4, we present the results obtained from our analysis. In Section 5, we discuss the implications of these results and the main threats to their validity. We relate them to previous work in the field in Section 6. In Section 7, we provide a summary of our findings and offer recommendations for future research.

A [replication package](#) is available on GitHub [3].

2 Background

2.1 Semantic versioning and Web API 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 backwards compatible bug fixes.

Several widely known package managers, such as NPM [4], Maven [21], 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 [12].

When it comes to Web APIs, in addition to informing about the version in the API metadata [13], clients may also refer to specific API version when

invoking them. The version identifier 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>/API-URL/<version-identifier>/` as well as a part of the server URL DNS name, such as: `<server-address> = v1.api.com | v2.api.com`. Embedding version identifiers in endpoint URLs is commonly used also when multiple versions of the API coexist simultaneously.

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 [20] or based on a standard Interface Description Language (IDL), such as OpenAPI [2]. OpenAPI 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 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 [6,7,15]. In our collection, 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 about 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.

SWR Audio Lab - Radiohub

Version: 2.14.0

Description: This documentation is also available as [openapi.json](https://docs.radiohub.swr.digital/openapi.json) or [openapi.yaml](https://docs.radiohub.swr.digital/openapi.yaml)

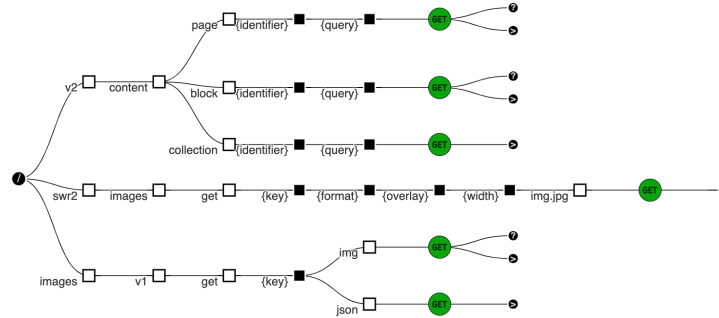


Fig. 1: Tree visualization of the endpoint structure of a subset of the SWR Audio Lab - Radiohub API (version 2.14.0) [8] (see the whole API tree) - Different version identifiers (v1, v2) are found in the path URL addresses.

– *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 pre-release versions are found.

3 Dataset Overview and Methodology

In this paper we analyze the versioning practices of Web APIs through an analysis of their descriptions written according to the OpenAPI specification [2]. Our dataset consists of 7,114 Web APIs, obtained from 186,259 commits pushed to 3,090 open-source GitHub repositories, belonging to 2,899 GitHub repository owners. To obtain this dataset, we filtered from 567,069 detected potential OpenAPI descriptions to include only those that: (a) belonged to APIs with more than 10 commits in their history (11,408 APIs); (b) were described in JSON/YAML files that were parsable in all commits (10,062); and (c) had at least one path specified in one of the commits (7,114), excluding descriptions of JSON schemas without any API functionality [18].

To automate the extraction, we first retrieved 5,514 distinct version identifiers from the `info.version` field in each OpenAPI description. We then searched for any of these identifiers in the URL addresses listed as part of the endpoints or server URL strings. For example, the Radiohub API [8] from the SWR audio lab includes the version identifier, `v2`, in the endpoint URLs, which reflects the major version of the API (Figure 1).

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

Table 1: Some detectors used to classify the version identifier formats

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 and stable versions of the APIs. The complete list can be found in the Appendix A. All machine-readable regular expression rules are included in the [replication package](#).

4 Results

4.1 Location of Version Identifiers in API Descriptions

In Table 2, we classify the dataset of APIs and commits into eight categories based on the locations of the API version identifiers. For completeness, we also include separate categories for APIs that lack version identifiers (L_{nover}) and APIs whose version identifier is discovered dynamically ($L_{dynamic}$).

We aggregate the commits of each API history as follows:

- $L_l^1 = \{api \in \mathbf{APIs}, \exists c \in \mathcal{C}_{api}, L_l(c)\}$: the set of APIs where there is at least one commit where the version identifiers are located in L_l .

- $L_l^* = \{api \in \mathbf{APIs}, \forall c \in \mathcal{C}_{api}, L_l(c)\}$: the set of APIs where in all the commits the version identifiers are located in L_l .

where $\mathcal{C}_{api} = \{c\}$ is the set of commits found during the history of the api , $L_l(c)$ indicates whether for commit c version identifiers are found in location $l \in \{ips, ps, ip, is, p, s, i, nover, dynamic\}$.

Static Versioning. The majority of APIs (4,445 - 62.5%) and commits (102,986 - 55%) has version identifiers located only in the `info.version` field of the API description metadata (L_i). The version identifiers were present in all of the server and path URLs, as well as in the `info.version` metadata (L_{ips}) for 453 commits belonging to 41 APIs. We did not observe any APIs that contained version identifiers in both path and server URLs but not in the `info.version` metadata field (L_{ps}).

	Location			#Commits		#APIs			
	info	version	path	server	Total	Identical	L_i^1	L_i^*	CV
L_{nover}	-	-	-	-	2,076		168	76 (45%)	92 (54%)
$L_{dynamic}$	-	-	-	-	5,985		220	129 (58%)	91 (41%)
L_i	x	-	-	-	102,986		5022	4445 (89%)	1236 (24%)
L_p	-	x	-	-	915		61	12 (20%)	25 (40%)
L_s	-	-	x	-	70		6	1 (16%)	1 (16%)
L_{ip}	x	x	-	-	61,010	36,441	1512	1139 (75%)	173 (11%)
L_{is}	x	-	x	-	18,749	4,050	1017	741 (73%)	93 (9%)
L_{ps}	-	x	x	-	0	0	0	0	0
L_{ips}	x	x	x	-	453	8	41	16 (39%)	5 (12%)
Total Statically Versioned					184,183	40499	7038	6354 (90%)	2390 (34%)

Table 2: Number of Commits/APIs which include version identifiers in different locations of the OpenAPI description artifacts

When version identifiers were present in multiple locations, we checked whether the identifiers were consistent or varied across those locations. We found that the identifiers were identical in 50.49% of the commits found in the history of 406 “consistently versioned” APIs. Furthermore, we identified 168 APIs that did not include any version identifiers in any location (L_{nover}) for certain commits, with 76 of these lacking all types of versioning throughout their entire history.

Dynamic Versioning. The version information of an API can also be obtained dynamically by the client via a dedicated API endpoint. Instead of specifying the version statically in the `info.version` field value, clients may retrieve the API version dynamically by invoking the `GET /version` operation, as documented in the example `{"version": "see /version below"}`. This approach was detected in 220 APIs in our collection, where 129 of them were dynamically versioned during their entire history, such as the [ONS](#) Address Index API.

4.2 Evolution of Version Identifiers

Given the API history $\mathcal{C}_{api} = \{c_i\}$, we define CV as the set of APIs where we detect at least one change in the value of the version identifier between two distinct commits:

$$CV = \{api \in \mathbf{APIs}, \exists c_j, c_i \in \mathcal{C}_{api}, version(c_j) \neq version(c_i)\}$$

For each L_l , in Table 2 we report as CV the number of APIs which change their version identifiers. About one third of the APIs (2,390) undergo at least one change of version identifier throughout their history of at least 10 commits.

In Figure 3, we present the correlation between the number of version changes in API histories and the number of commits, differentiated by the location of the version identifier. The dot color indicates the number of APIs with a given

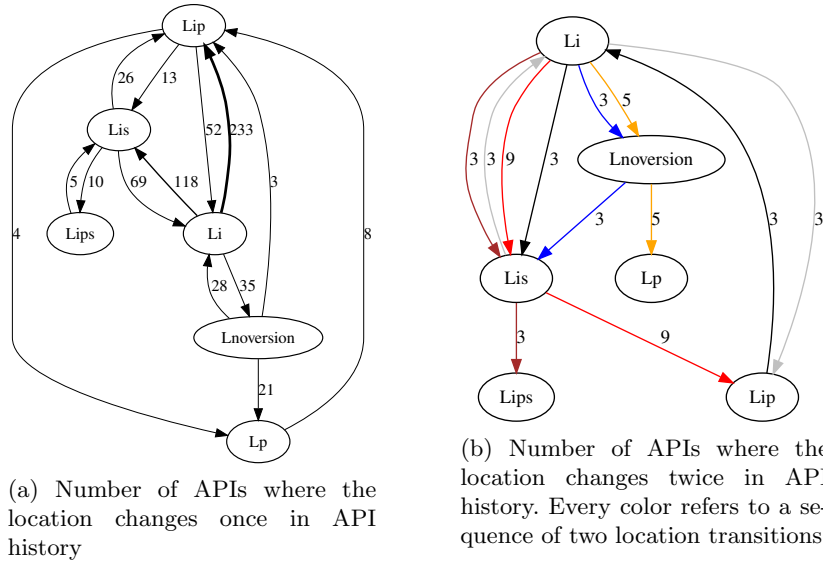


Fig. 2: Number of APIs where the version identifiers changes its location during the API’s history (includes only the changes observed in at least two APIs)

combination of version changes (x) and commits (y). As anticipated, $y > x$. A small subset of APIs (47 APIs) demonstrates frequent version changes with each commit ($y = x + 1$), while a considerable number of APIs (L_i : 5119, L_p : 1446, L_s : 973) maintain a constant version identifier ($x = 0, y \geq 10$) despite having in some cases a substantial number of commits in their history. The majority of version changes occur in the location designated as L_i , with fewer changes observed for version identifiers embedded in URLs.

By analyzing changes of the `info.version` field, we could track the API evolution through several iterations of preview releases followed by stable releases (and vice-versa). Figure 4 shows how many APIs evolved with preview and stable releases, and how many added or removed versioning information at some commit of their history. We also measured the delay between each type of release: on average, preview releases occur every 9.3 days, while stable versions are released every 18.2 days.

4.3 Classification of Version Identifier Formats

We categorize the formats employed to represent version identifiers in Table 3. The results show that the most widely utilized format for versioning API releases is Semantic Versioning (SemVer), followed by a straightforward approach using an integer to denote the major version of the API, often accompanied by a V prefix. All types of preview release tags are most often found in the `info.version` metadata, while release candidate and preview tags are never found as part of path or server URLs.

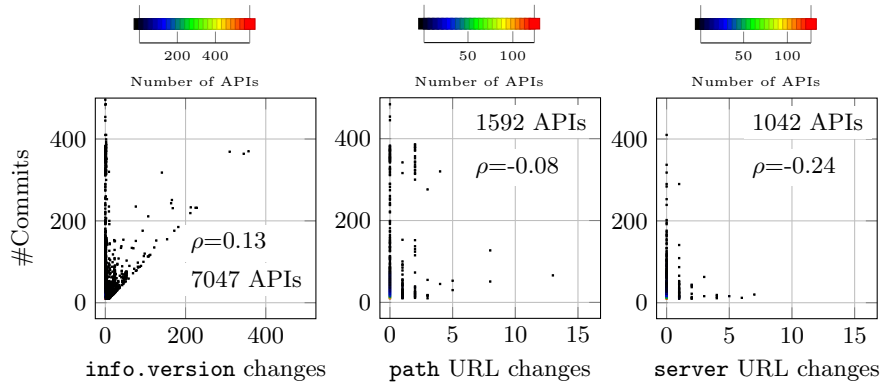


Fig. 3: Where do version identifiers change more often? Density plots of the number of commits of each API as a function of the number of version identifier changes detected in the three locations (L_i, L_p, L_s).

The version formats for 534 out of 7114 APIs exhibited instability over time, with the most prevalent change being from the absence of a version to Semantic Versioning (SemVer). The most common format transitions are listed in Figure 5. In contrast, 4941 APIs consistently utilized SemVer throughout their history. Table 4 provides a more detailed classification of the version identifier format used by APIs that maintained a consistent versioning scheme across all commits, including statistics on the number of commits and version identifier changes. We also include the most frequent version identifier detected in each category.

4.4 Adoption of Semantic Versioning Over the Years

In our collection of 7,114 APIs, we discovered that 5,292 APIs employed semantic versioning (with 112,908 commits) at some point in their history. Additionally, 504 APIs utilized alternative formats in addition to semantic versioning (with 7,388 commits) and 4,565 APIs exclusively utilized SemVer-3 (e.g., 1.0.0) and SemVer-2 (e.g., 1.0) throughout their history.

An examination of the adoption of semantic versioning (SemVer) over time, as depicted in Figure 6, reveals that the usage of SemVer in final release versions is consistently higher than other versioning schemes, with a relatively steady level of adoption from 2015 to 2021.

4.5 Multiple Versions in Production

Our analysis identified a total of 135 APIs that incorporate various version identifiers in their paths, as demonstrated in Figure 7 (a). Out of these, 2102 paths had two distinct version identifiers, while one API reached a maximum of 14 co-existent versions during 5 commits. In addition, 51.11% (69 APIs) demonstrated a change in the number of path versions at least once.

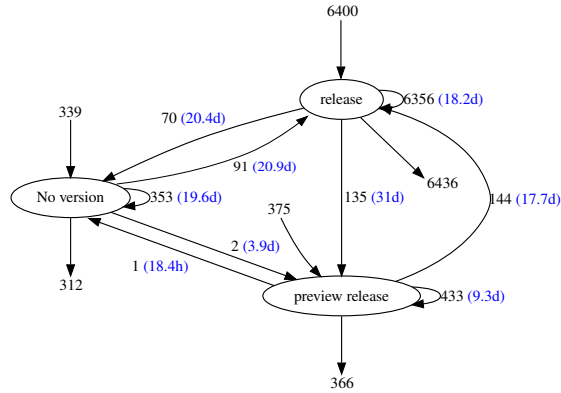


Fig. 4: Number of APIs which evolve interleaving preview and stable releases and average duration (in days) of the transitions

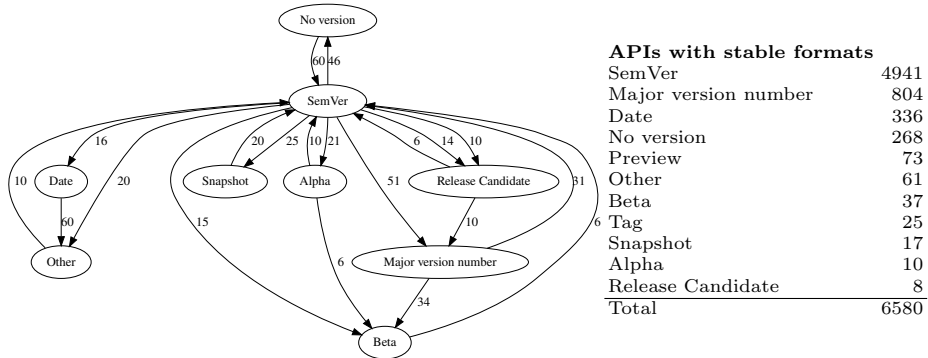


Fig. 5: Number of APIs where `info.version` identifiers formats changes were detected during their history (includes only transitions happening in ≥ 4 APIs)

As an example, the RiteKit API introduced a new path with a `v2` version identifier. The Agent API underwent two changes: from `v1, v3, v4` to `v1, v2, v3, v4` on 2017-08-02, and from `v1, v2, v3, v4` to `v1, v2, v3, v4, v5` on 2017-09-27. Additionally, the version declared in the metadata increased from `1.46.0` to `22.9.1` over 51 commits during a five-year period, showing an example of inconsistency between the version identifiers found in the API description metadata and the endpoints exposed to the API clients.

We categorize the version identifier formats present in APIs with multiple versions in production in Figure 7. The predominant trend among APIs with multiple versions is to adopt a similar version format consisting of the use of a major version number (MVN) often attached to the prefix `V*`, especially in APIs that have fewer than six concurrent versions. We observed a deviation from this trend in only 593 commits, out of which 292 commits combined different version formats, as depicted in Figure 7 (b). This deviation is more commonly seen in

Format	Location			
	info.version	path	server	All
Major version number	29129	45310	14944	89383
SemVer	114663	788	18172	133623
Tag	845	1199	6	2050
Date	5447	299	27	5773
Other	1549	1354	0	2309
Develop	545	92	106	743
Snapshot	964	0	11	975
Preview	863	0	0	863
Alpha	3003	2339	10	5352
Beta	19410	15459	207	35076
Release Candidate	548	0	0	548

Table 3: Number of commits with version identifiers of stable (above) and preview (below) releases classified by their format and location (a more detailed classification is in Appendix B)

APIs that have more than six concurrent versions, and the most prevalent format among these is the Semantic Versioning (SemVer) format.

5 Discussion

Q1: *What is the prevalence of versioning in Web APIs? How often is version information located outside of the API metadata or discovered dynamically?*

Our study found that out of the 7114 APIs examined, only 76 were completely unversioned, and 336 started their history with no metadata version while in [15] the authors recommend using a version number from the start of the lifespan of any software artifact. Of the remaining APIs, 4445 included static version information exclusively in their metadata, while the others had version identifiers present in their Paths or Server URLs. Notably, in the case of 1896 APIs, version identifiers were present both in the `info.version` field and endpoint URLs. Conversely, none of the APIs had version identifiers both in Paths and Server URLs, without also having it in the `info.version` field. Another versioning practice was detected in the case of 220 APIs where the developers dedicated one of the endpoints to dynamically inform clients about the version of the currently deployed API.

Q2: *How do developers distinguish stable from preview releases?*

Our analysis identified specific labels indicating a different type of preview release version in 25,308 commits belonging to 535 APIs. The examination of 202 APIs revealed that they have a history covering both preview and stable releases. The absolute number and the portion of pre-releases increases during recent years (Figure 6). The labeling accuracy appears to be confirmed also by the differences in the average delay measured for each type of release (Figure 4).

Format	Most Frequent Version Identifier			#Commits				VC			
	Version Identifier	#APIs		max	avg	mdn	stdev	max	avg	mdn	stdev
semver-3	1.0.0	40.45%	3531	1031	28	17	37	496	4	0	17
semver-2	1.0	64.92%	1093	3585	30	15	116	77	1	0	4
v*	v1	80.32%	489	692	42	20	74	4	0	0	0
date(yyyy-mm-dd)	2017-03-01	4.87%	327	52	14	12	4	52	0	0	3
other	v1b3	7.23%	213	222	29	18	32	33	1	0	3
integer	1	36.30%	48	143	27	17	24	113	5	0	20
v*beta*	v1beta1	60.10%	115	360	136	35	146	3	0	0	1
date-preview*	2015-10-01-preview	11.93%	72	47	13	12	5	2	0	0	0
semver-3#	1.0.0-oas3	27.62%	33	215	32	15	41	18	2	0	4
v*beta*.*	v2beta1.1	19.44%	26	30	24	24	4	12	3	3	4
latest*	latest	52.75%	25	137	27	15	28	2	0	0	0
v*alpha*	v1alpha	51.34%	18	339	56	24	91	3	0	0	1
semver-SNAPSHOT*	1.0.0-SNAPSHOT	31.61%	18	172	32	16	38	36	5	0	9
semver-beta*	v1.0-beta	28.37%	17	113	40	29	29	9	1	0	2
v*p*beta*	v1p3beta1	23.45%	9	347	162	35	153	3	1	0	1
beta	1beta1	100.00%	7	37	15	11	9	0	0	0	0
beta*	beta	65.49%	7	47	26	26	12	0	0	0	0
semver-alpha*	1.0.0-alpha	28.04%	7	48	23	15	15	2	0	0	1
semver-2#	1.3-DUMMY	12.26%	6	24	16	15	5	3	2	2	1
semver (beta*)	1.0 (beta)	29.89%	6	58	39	46	13	46	18	26	18
date(yyyy.mm.dd)	2019.10.15	10.45%	6	24	22	24	4	24	20	24	9
#semver-3	2019.0.0	29.73%	5	37	22	17	11	3	1	2	1
semver-rc*	1.0.0-rc1	38.14%	4	190	60	20	75	8	4	5	3
semver-4	6.4.3.0	3.31%	4	23	16	17	5	9	2	0	4
semver-rc*.*	2.0.0-RC1.0	41.69%	4	85	54	63	26	0	0	0	0
v*alpha*.*	v2alpha2.6	61.76%	3	26	23	22	2	4	1	0	2
alpha*	alpha	73.85%	2	35	26	35	9	0	0	0	0
dev*	dev	98.38%	2	172	91	172	81	0	0	0	0
date(yyyy-mm)	2021-10	67.44%	2	14	13	14	1	2	1	2	1
semver-pre*.*	3.5.0-pre.0	100.00%	1	10	10	10	0	0	0	0	0
date(yyyymmdd)	20190111	29.63%	1	13	13	13	0	0	0	0	0
semver-dev*	0.7.0.dev20191230	15.52%	1	40	40	40	0	0	0	0	0
v*-date	v1-20160622	57.14%	1	18	18	18	0	2	2	2	0
semver-alpha*.*	1.1.0-alpha.1	4.94%	1	146	146	146	0	0	0	0	0

Table 4: How many APIs consistently use the same version identifier format in the `info.version` field during their lifespans? $\min(\#Commits) = 10$, $\min(\text{info.version Changes}) = 0$

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

Our findings indicate that semantic versioning (SemVer) is the most widely adopted versioning scheme among API releases, specifically in the case of APIs that use the `info.version` field (60.56%). In contrast, for APIs that use alternative methods for versioning, such as embedding version identifiers in endpoints URLs or the server DNS name, the most common practice is to use shorter version identifiers that include only the major version counter. Additionally, for preview releases, SemVer is often combined with other tags to reflect the type of preview release, such as Develop, Snapshot, Alpha, Beta, or Preview (Table 8). More in detail, 3-counter semantic versioning (MAJOR.MINOR.PATCH) is adopted more often than the 2-counter format (BREAKING.NONBREAKING) recommended by [13]. The fourth largest group of APIs uses “visible dates” as version identifier

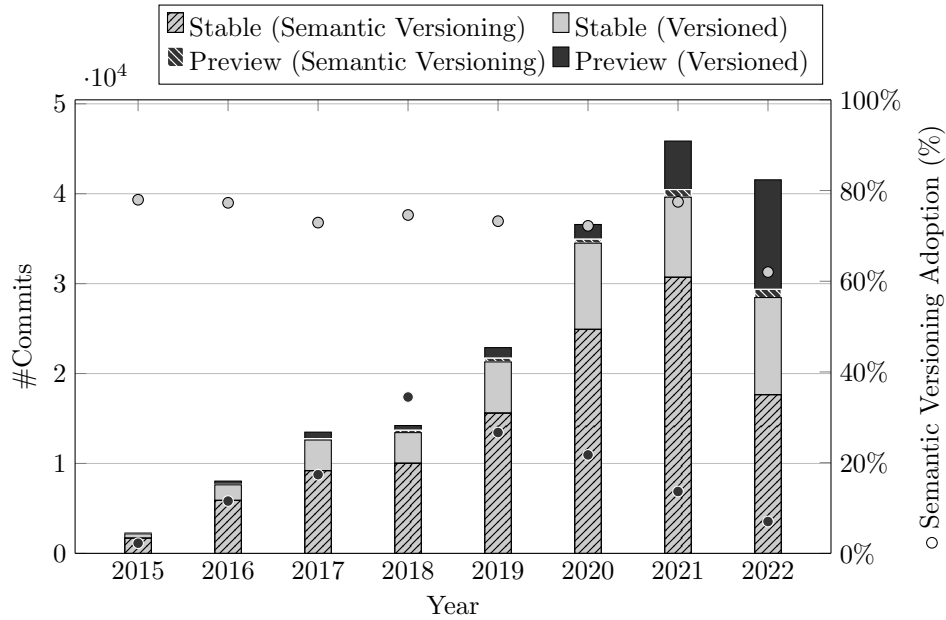


Fig. 6: Semantic versioning adoption over the years (2015-2022) in stable and preview releases considering the version identifier found in the `info.version` metadata.

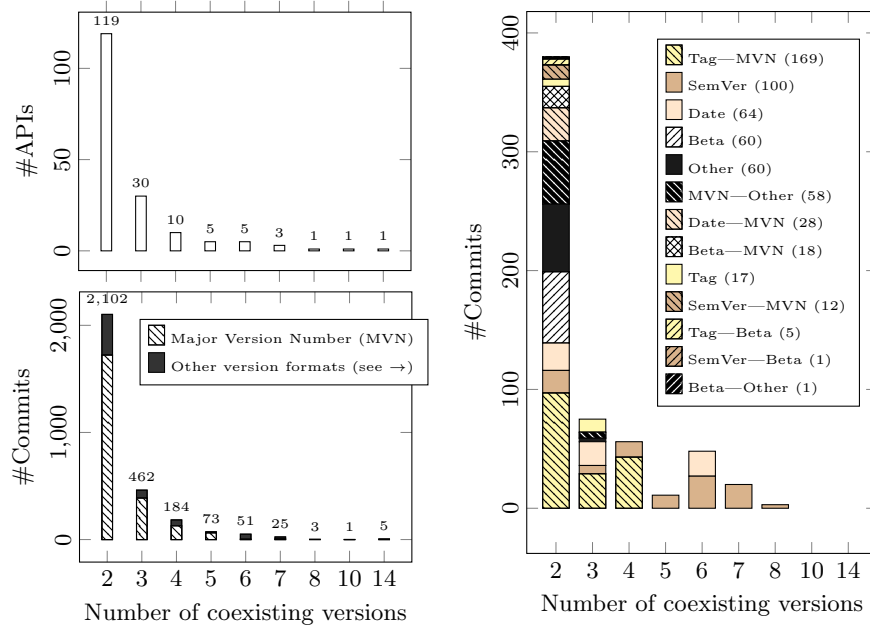
format (Table 4), which appears to be anti-pattern according to [15], as it would reveal how old a release has become.

Q4: *How often do developers switch to different versioning schemes during the lifespan of their APIs?*

Only 785 APIs underwent changes in their versioning scheme during their evolution. Figure 5 illustrates that, in the case of APIs that use `info.version` identifiers, the most common target format adopted by 106 APIs changing their scheme is Semantic Versioning (SemVer), which is also the most widely adopted versioning scheme among APIs that did not switch to another format.

Q5: *Has the adoption of semantic versioning changed over the past few years?*

In this study, we assessed the prevalence of Semantic Versioning (SemVer) in API versioning by analyzing the utilization of the `info.version` field in stable releases of APIs that have been committed to GitHub between 2015 and 2022. Our findings showed a relatively high adoption rate of SemVer in stable releases, with a mean of $75.84\% \pm 4.79\%$. However, the adoption rate was lower for API preview releases, where the most commonly used formats did not conform to the SemVer format (Table 8). Our analysis revealed a linear decline in the adoption of SemVer in preview releases from 2018 to 2022, with a significant increase in the use of simpler versioning formats, such as `v*beta*` or `v*alpha*`, which combine the major version number with a preview release tag.



(a) Number of API/Commits having paths indicating more than one version (b) Other version formats used in commits with n-coexistent versions

Fig. 7: Multiple Versions in Production

Q6: *What is the prevalence of APIs with multiple versions in production? how many concurrent versions exist, and which formats are used in this case?*

We analyzed the usage of the “two in production” evolution pattern [14,22] by examining the APIs that have paths with distinct version identifiers. Out of 7114 APIs, we found 175 with multiple version identifiers attached to their paths. The majority (119 APIs) had exactly two versions across 2,102 commits. In one case, an API supported up to 14 different versions in production across five commits. The commonly used format for version identifiers attached to the API path varies based on the number of coexisting versions. For APIs with fewer than six versions, the most prevalent format is to reference only the major version. However, for APIs with more than six coexisting versions, the widely adopted format is semantic versioning. This suggests that an increased number of coexisting versions requires a more detailed identifier for proper differentiation.

5.1 Threats to Validity

- Internal validity: Our formats classification approach allowed us to accurately identify the presence of version identifiers embedded in the URL. However, this process is susceptible to errors or omissions if the version information is not

already retrieved from the `info.version` or if the identifier matched in the URL is not meant to be used for versioning.

- External validity: Our study relies on API history snapshots from GitHub, potentially missing the full evolution of the APIs. Findings should be viewed with caution given the possibility of missing updates and changes. The APIs in the study come from a single platform, GitHub, and caution should be exercised in generalizing the results to APIs developed elsewhere.

6 Related Work

In previous work on Web API evolution [11] 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 [21,16], 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. [12], 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 [10], 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.

7 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 location, formats, and evolution. Out of 7114 APIs, the majority (5022) utilized static versioning in the API metadata, while a small

fraction (220) supported dynamic discovery of the current version through a dedicated endpoint. In terms of version format, we identified 55 distinct formats used to distinguish stable and preview releases, with 535 APIs including preview versions in their Github histories. The number of preview releases pushed to Github showed an upward trend with a yearly average of 1858 commits.

With regards to metadata versions, we found that 85% of the 6580 APIs which consistently used the same format throughout their lifespan utilized Semantic Versioning. The adopted version format was unstable in 534 APIs, with 30% switching to SemVer. Our analysis indicated a steady usage rate of SemVer for 75% (on average) of API releases, while pre-releases adopted more often less detailed formats that only reference the major version of the API, typically with a tag (e.g., “beta” being the most frequent) to indicate their purpose.

We also observed the usage of the “two in production” evolution pattern in 175 APIs (56 with more than 2 versions). In these cases, the most prevalent format for version identifiers attached to the path was to reference only the major version, particularly among APIs with fewer than six coexisting versions.

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 The authors acknowledge Fabio Di Lauro for gathering the raw dataset, and Deepansha Chowdhary for conducting a feasibility study on it. This work was supported by the SNF with the API-ACE project number 184692.

References

1. Semantic Versioning. <https://semver.org/>
2. OpenAPI Initiative. <https://www.openapis.org/>
3. <https://github.com/USI-INF-Software/API-Versioning-practices-detection>
4. <https://docs.npmjs.com/about-semantic-versioning>
5. SwaggerHub. <https://app.swaggerhub.com/>
6. Release naming conventions. <https://www.drupal.org/node/1015226>
7. <https://docs.fedoraproject.org/en-US/packaging-guidelines/Versioning/>
8. SWR Audio Lab - Radiohub API. <https://github.com/swrlab/swrlab/blob/main/openapi/openapi.yaml>
9. Bogart, C., Kästner, C., Herbsleb, J., Thung, F.: How to break an API: cost negotiation and community values in three software ecosystems. In: Proc. 24th International Symposium on Foundations of Software Engineering. pp. 109–120 (2016)
10. Decan, A., Mens, T., Mazrae, P.R., Golzadeh, M.: On the use of github actions in software development repositories. In: International Conference on Software Maintenance and Evolution (ICSME). pp. 235–245 (2022)
11. Di Lauro, F., Serbout, S., Pautasso, C.: A large-scale empirical assessment of web api size evolution. *Journal of Web Engineering* **21**(6), 1937–1980 (2022)
12. Dietrich, J., Pearce, D., Stringer, J., Tahir, A., Blincoe, K.: Dependency versioning in the wild. In: Proc. 16th International Conference on Mining Software Repositories (MSR). pp. 349–359 (2019)

13. Lauret, A.: The design of web APIs. Simon and Schuster (2019)
14. Lübke, D., Zimmermann, O., Pautasso, C., Zdun, U., Stocker, M.: Interface evolution patterns: balancing compatibility and extensibility across service life cycles. In: Proc. 24th EuroPLoP (2019)
15. Marquardt, K.: Patterns for software release versioning. In: Proc. of the 15th European Conference on Pattern Languages of Programs (EuroPLoP) (2010)
16. Ochoa, L., Degueule, T., Falleri, J.R., Vinju, J.: Breaking bad? semantic versioning and impact of breaking changes in maven central. *Empirical Software Engineering* **27**(3), 1–42 (2022)
17. Raatikainen, M., Kettunen, E., Salonen, A., Komssi, M., Mikkonen, T., Lehtonen, T.: State of the practice in application programming interfaces (APIs): A case study. In: Proc. 15th European Conference on Software Architecture (ECSA). pp. 191–206 (2021)
18. Serbout, S., Di Lauro, F., Pautasso, C.: Web apis structures and data models analysis. In: Companion Proc. 19th International Conference on Software Architecture (ICSA). pp. 84–91 (2022)
19. Varga, E.: Creating Maintainable APIs. APress (2016)
20. Yang, J., Wittern, E., Ying, A.T., Dolby, J., Tan, L.: Towards extracting web api specifications from documentation. In: Proceedings of the 15th International Conference on Mining Software Repositories (MSR). pp. 454–464 (2018)
21. Zhang, L., Liu, C., Xu, Z., Chen, S., Fan, L., Chen, B., Liu, Y.: Has my release disobeyed semantic versioning? static detection based on semantic differencing. In: Proc. 37th International Conference on Automated Software Engineering (2023)
22. Zimmermann, O., Stocker, M., Lübke, D., Zdun, U., Pautasso, C.: Patterns for API Design - Simplifying Integration with Loosely Coupled Message Exchanges. Addison-Wesley (2022)

A Version Formats Detectors

This appendix lists only the detectors (stable releases: Table 5 and preview releases: Table 6) that enabled classifying the most frequently occurring formats listed in Tables 7 and 8 of Appendix B. The dataset and the detectors code can be found in the replication package on GitHub [3].

Format	Regex detector
Major version number	
integer	<code>integer: /^(\\d{3} \\d{2} \\d{1})+\$/i</code>
<code>v*</code>	<code>/^v\\d\$/i</code>
SemVer	
<code>#semver-2</code>	<code>/(v)(\\d{3} \\d{2} \\d{1})\\.\\d+\$/i</code>
<code>#semver-3</code>	<code>/(v)(\\d{3} \\d{2} \\d{1})\\.\\d+\\.\\d+\$/i</code>
<code>#semver-6</code>	<code>/(v)\\d+\\.\\d+\\.\\d+\\.\\d+\\.\\d+\\.\\d+\$/i</code>
<code>semver-2</code>	<code>/^ (v) (\\d{3} \\d{2} \\d{1}) \\. \\d+\$/i</code>
<code>semver-2#</code>	<code>/^ (v) (\\d{3} \\d{2} \\d{1}) \\. \\d+\$/i</code>
<code>semver-3</code>	<code>/(v)(\\d{3} \\d{2} \\d{1})\\.\\d+\\.\\d+\$/i</code>
<code>semver-3#</code>	<code>/(v)(\\d{3} \\d{2} \\d{1})\\.\\d+\\.\\d+\$/i</code>
<code>semver-4</code>	<code>/(v)\\d+\\.\\d+\\.\\d+\\.\\d+\$/i</code>
<code>semver-5</code>	<code>/(v)\\d+\\.\\d+\\.\\d+\\.\\d+\\.\\d+\$/i</code>
<code>semver-6</code>	<code>/(v)\\d+\\.\\d+\\.\\d+\\.\\d+\\.\\d+\\.\\d+\$/i</code>
<code>semver-6#</code>	<code>/(v)\\d+\\.\\d+\\.\\d+\\.\\d+\\.\\d+\\.\\d+\$/i</code>
Tag	
<code>stable*</code>	<code>/^stable\\d*\$/i</code>
<code>latest*</code>	<code>/latest\\d*\$/i</code>
Date	
<code>date(Month yy)</code>	<code>/(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec) [1-2] [0-9]\$/i</code>
<code>date(Month yyyy)</code>	<code>/(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec) 20[1-2] [0-9]\$/i</code>
<code>date(yyyy)</code>	<code>/^\\d{4}\$/i</code>
<code>date(yyyy-mm)</code>	<code>/^\\d{4}-\\d{2}\$/i</code>
<code>date(yyyy-mm-dd)</code>	<code>/^\\d{4}-\\d{2}-\\d{2}\$/i</code>
<code>date(yyyy.mm)</code>	<code>/(^\\d{4}\\. [1-12]\$/i ^\\d{4}\\. \\d{2}\$/i)</code>
<code>date(yyyy.mm.dd)</code>	<code>/\\d{4}\\. \\d{2}\\. \\d{2}\$/i</code>
<code>date(yyyy/mm)</code>	<code>/^\\d{4}\\. \\d{2}\$/i</code>
<code>date(yyyy/mm/dd)</code>	<code>/^\\d{4}\\. \\d{2}\\. \\d{2}\$/i</code>
<code>date(yyyymmdd)</code>	<code>/^20[1-2] [0-9] (0[1-9] 1[0-2]) (0[1-9] [12] [0-9] 3[01])\$/i</code>
Other	Any other format that wasn't caught by one of the detectors defined in here

Table 5: Detectors of version identifiers formats of API releases. * stands for an integer and # stands for an arbitrary combination of characters

Format	
Develop	
dev*	/^dev\d*\$/i
develop*	/^develop\d*\$/i
semver-dev*	/^(v)\d+\.\d+(\.\d)*(\. - _)dev\d*\$/i
semver-dev*.*	/^(v)\d+\.\d+(\.\d)*(\. - _)dev\d*\.\d+\$/i
v*dev*	/^v\d+dev\d*\$/i
Snapshot	
snapshot*	/^SNAPSHOT\d*\$/i
semver-SNAPSHOT*	/^(v)\d+\.\d+(\.\d)*(\. - _)SNAPSHOT\d*\$/i
Preview	
date-preview*	[date](- \.)preview\$/i
semver-pre*.*	/^(v)\d+\.\d+(\.\d)*(\. - _)pre\d*\.\d+\$/i
semver-preview*	/^(v)\d+\.\d+(\.\d)*(\. - _)preview\d*\.\d+\$/i
Alpha	
alpha*	/^alpha\d*\$/i
semver-alpha*	/^(v)\d+\.\d+(\.\d)*(\. - _)alpha\d*\$/i
semver-alpha*.*	/^(v)\d+\.\d+\.\d+(\.\d)*(\. - _)alpha\d*\.\d+\$/i
v*alpha*	/^v\d+alpha\d*\$/i
v*alpha*.*	/^v\d+alpha\d+\.\d+\$/i
Beta	
beta	/^\d+beta\d*\$/i
beta*	/^beta\d*\$/i
semver (beta*)	/^(v)\d+\.\d+(\.\d)*(\. - _)\(beta\d*\)/i
semver beta*	/^(v)\d+\.\d+(\.\d)*(\. - _)beta\d*\$/i
semver.beta*.date	/^(v)\d+\.\d+(\.\d)*(\. - _)beta\d*(\. - _)\d{4}\.\d{2}\.\d{2}\$/i
v*.beta	/^v\d+(\.\d)*beta\d*\$/i
v*beta*	/^v\d+beta\d*\$/i
v*beta*.*	/^v\d+beta\d+\.\d+\$/i
v*p*beta*	/v\d+p\d+beta\d*\$/i
Release Candidate	
rc*	/^rc\d*\$/i
semver-rc*	/^(v)\d+\.\d+(\.\d)*(\. - _)rc\d*\$/i
semver-rc*.*	/^(v)\d+\.\d+(\.\d)*(\. - _)rc\d*\.\d+\$/i
semver.rc*.date	/^(v)\d+\.\d+(\.\d)*(\. - _)rc\d*(\. - _)[date]\$/i

Table 6: Detectors of version identifiers formats of API preview releases. * stands for an integer

B Versions Formats Classification

In this appendix we include detailed results about the version identifier classification for stable (Table 7) and preview (Table 8) releases.

Format	Location			
	info.version	path	server	All
Major version number	29129	45310	14944	89383
integer	3738	592	144	4474
v*	25391	44718	14800	84909
SemVer	114663	788	18172	133623
#semver-2	166	0	4249	4415
#semver-3	111	0	2686	2797
#semver-6	6	0	0	6
semver-2	23248	772	1594	25614
semver-2#	212	0	4260	4472
semver-3	88964	22	2686	91672
semver-3#	1448	0	2697	4145
semver-4	513	3	3	519
semver-5	1	3	3	7
semver-6	20	3	3	26
semver-6#	2	0	0	2
Tag	845	1199	6	2050
latest*	819	1174	3	1996
stable*	26	25	3	54
Date	5447	299	27	5773
date(Month yy)	0	3	3	6
date(Month yyyy)	22	0	0	22
date(yyyy)	21	3	3	27
date(yyyy-mm)	43	33	3	79
date(yyyy-mm-dd)	5146	245	3	5394
date(yyyy.m)	10	0	0	10
date(yyyy.mm)	4	3	3	10
date(yyyy.mm.dd)	134	3	3	140
date(yyyy/mm)	20	3	3	26
date(yyyy/mm/dd)	20	3	3	26
date(yyyymmdd)	27	3	3	33
Other	1549	1354	0	2309

Table 7: Number of commits with version identifiers of API releases classified by their formats. * stands for an integer and # stands for an arbitrary combination of characters

Format	Location			All
	info.version	path	server	
Develop	545	92	106	743
dev*	185	91	104	380
develop*	12	0	2	14
semver-dev*	335	0	0	335
semver-dev*.*	12	0	0	12
v*dev*	1	1	0	2
Snapshot	964	0	11	975
SNAPSHOT*	34	0	0	34
semver-SNAPSHOT*	930	0	11	941
Preview	863	0	0	863
date-preview*	830	0	0	830
semver-pre*.*	10	0	0	10
semver-preview*	23	0	0	23
Alpha	3003	2339	10	5352
alpha*	65	25	10	100
semver-alpha*	510	0	0	510
semver-alpha*.*	385	0	0	385
v*alpha*	1975	2314	0	4289
v*alpha*.*	68	0	0	68
Beta	19410	15459	207	35076
beta	105	0	0	105
beta*	339	37	175	551
semver (beta*)	368	0	0	368
semver beta*	557	0	0	557
semver.beta*.date	9	0	0	9
v*.beta	254	44	0	298
v*beta*	15720	15378	32	31130
v*beta*.*	638	0	0	638
v*p*beta*	1420	0	0	1420
Release Candidate	548	0	0	548
rc*	26	0	0	26
semver-rc*	118	0	0	118
semver-rc*.*	355	0	0	355
semver.rc*.date	49	0	0	49

Table 8: Number of commits with version identifiers of API preview releases classified by their formats. * stands for an integer