How Many Web APIs Evolve Following Semantic Versioning?

Souhaila Serbout [ID] and Cesare Pautasso [ID]

Software Institute (USI), Lugano, Switzerland
souhaila.serbout@usi.ch, c.pautasso@ieee.org

Abstract. More and more Web APIs use semantic versioning to represent the impact of changes on clients depending on previous versions. Our goal is to provide insights about the extent to which evolving Web APIs align with semantic versioning rules. In this paper we present the results of an empirical study on the descriptions of 3 075 Web APIs, which released at least one new version throughout their history. The APIs descriptions were mined by retrieving 132 909 commits from 2 028 different open source GitHub repositories. We systematically collected and examined 506 273 changes of 195 different types released within 16 053 new API versions. We classified whether each change is likely to break clients or not, and checked whether the corresponding version identifier has been updated following semantic versioning rules. The results indicate that in the best case, only 517 APIs consistently release major upgrades when introducing breaking changes, while 1 970 APIs will not always correctly inform their clients about breaking changes released as part of minor or patch-level upgrades. We also detected 927 APIs which use a backwards-compatible evolution strategy, as they never introduce any breaking change throughout their history.

Keywords: Web APIs, Semantic Versioning, API Evolution, Breaking Changes, OpenAPI

1 Introduction

In the rapidly evolving landscape of software development, Application Programming Interfaces (APIs) stand as critical components [6], facilitating seamless interactions between different software systems and services [23,15]. The management of API evolution through versioning makes it possible to check, ensure or break compatibility and determine how changes will affect API clients [10,8,12]. Semantic Versioning (SemVer) has emerged as a widely adopted set of rules aimed at clarifying how to mint version identifiers to describe the impact of changes on clients depending on previous versions of an API [2]. The adoption and compliance with Semantic Versioning has been empirically studied [4] within repositories of software packages and libraries for different programming languages (e.g., Maven [17,13], npm [16], golang [9]). Despite its prevalence within

Web API descriptions [19], there is a lack of empirical studies on the adherence to and the correct usage of semantic versioning [1] in real-world Web APIs.

In this paper, we aim at bridging this gap by presenting a method to assess the consistency between changes applied to OpenAPI descriptions and the corresponding version identifier which leads to answering the following:

RQ1) How often APIs introduce breaking vs. non-breaking changes?

RQ2) Are there many Web APIs which consistently follow semantic versioning rules across their entire history?

Given the public nature of Web APIs, the expectation is that their developers carefully assess the impact of every change as they strive to avoid breaking their clients. But if breaking changes are introduced, are semantic versioning rules properly followed? How often can clients rely on semantic versioning identifiers to set their expectations about the impact of new releases they depend on?

In this paper we present a data analysis method to statically classify 195 different types of changes that can be detected by comparing OpenAPI [14] descriptions and predict whether they are likely to break clients with different tolerance levels [3]. We apply the method to a collection of 3 075 API evolution histories mined from open source GitHub repositories. The main findings are that, in the best case, 1) almost one third of APIs in our sample (927) evolves in a backwards compatible way; 2) a minority of APIs (517) which introduce breaking changes does so by consistently adhering to semantic versioning rules.

The rest of this paper is outlined as follows: In Section 2, we highlight the principal studies related to our work. Section 3 details the dataset of OpenAPI specifications analyzed in this study. In Section 4, we introduce the key definitions utilized throughout the paper. The methodology adopted for our analysis, along with the metrics calculated, are elaborated in Section 5, with the findings presented in Section 6. Discussions on the implications and the validity threats of these results are found in Section 7. Finally, Section 8 concludes the paper, outlining our conclusions and directions for future research.

2   Related Work

The consistent adoption of semantic versioning has been studied empirically for software packages released in programming languages like Maven [17,13], npm [16], golang [9]. Analyzing a large dataset from GitHub comprising 124k third-party golang libraries and 532k client programs, the authors of [9] found that 86% of the golang libraries follow semantic versioning but 28.6% of non-major releases introduced breaking changes.

In [17], the authors scrutinize semantic versioning compliance in the Maven repository, analyzing over 10 000 `.jar` files from 22 000 libraries. It uncovers that 33% of releases breach semantic versioning by introducing breaking changes, which deviates from the expected practice of only making breaking changes in major releases. In a more recent replication study [13], the authors analyzed 119 879 Java library upgrades and 293 817 clients revealing that 83.4% of upgrades adhere to semantic versioning.

As public Web APIs are meant to be offered to an unknown set of clients [23] – unlike the previous studies – we do not consider client-side artifacts or usage logs to estimate the impact of changes. Our static analysis therefore produces a conservative assessment on the impact of the detected API changes on clients.

In our previous research on Web API versioning practices [19], we analyzed 7 114 APIs from GitHub, revealing 55 different version formats. We found that 85% of these APIs consistently adopt identifiers syntactically consistent with semantic versioning. In further work [18], we proposed "API Version Clock" a visualization of the evolution of an API over time, emphasizing the relationship between changes of version identifiers and the nature of the changes made (e.g., breaking or non-breaking changes). It employs a sunburst plot to provide a fine-grained, chronological view of API changes, color-coded to distinguish between major, minor, and patch releases. Observing a small gallery of API evolution histories reveals widely different approaches to versioning decisions in response to changes. Building on these preliminary results, in this paper our objective is to systematically and quantitatively assess the consistency between the changes made in Web APIs and whether the corresponding version identifiers have been updated following the actual semantics of semantic versioning [1].

3   Dataset

The OpenAPI Specifications (OAS [14]) analyzed in this study were gathered through the GitHub API [20]. Before filtering, this dataset included 915 885 valid specifications from 270 578 APIs committed to GitHub between 2015 and January 2024. As described in Table 1, our analysis focuses on the evolutionary aspect of APIs. Therefore, we specifically looked at APIs with a history of at least 10 commits, all containing valid OAS documents. Considering the goal of this study, to examine the practical adoption of semantic versioning, we filtered for APIs that consistently use identifiers compatible with semantic versioning throughout their entire history. Additionally, to be able to check the level of compliance with semantic versioning rules, we identified APIs that have released at least one new version during their history that included some modifications impacting the functionalities of the API. As a result, our study includes the history of 3 075 APIs, with a total of 15 856 versions, corresponding to 506 273 changes introduced in their documentation.

Table 1: Data cleaning steps

Filtering Step	# APIs	# Commits
all valid commits	270 578	915 885
at least 10 valid commits	16 401	490 526
always use semantic versioning identifiers	14 489	413 463
have at least one version change	3 075	132 909

4   Definitions

4.1   Semantic Versioning Change Classification

Due to the lack of widely accepted semantics for arbitrary version identifiers, in this paper we focus exclusively on APIs which make consistent use of semantic versioning throughout their evolution history, in both stable and preview releases. More precisely, we analyzed API descriptions versioned with four different schemes: `X` (Major), `X.Y` (Major.Minor), `X.Y.Z` (Major.Minor.Patch), and `X.Y.Z-LABEL` (Major.Minor.Patch-Release Type). Where the release type, if present, labels the maturity of the artifact along the API release lifecycle. The version identifiers have been matched with the following regular expression:

`/^(?i)(v)?\d{1,3}(?:\.\d{1,3})?(?:\.\d{1,3})?(?:-LABEL ))?$/`

Where `LABEL` can be: alpha, beta, dev, snapshot, rc, preview, test, private.

We limit the size of the numbers to three digits because we want to avoid catching identifiers using dates [5], which are often used in versioning but do not provide a clear, incremental progression of versions, reflecting the expected change impact between releases.

Based on the previous regular expression, the parsing operation p transforms a version string into a structured tuple $(X, Y, Z, Label)$. E.g,: $p(\texttt{v1}) \rightarrow (1,0,0,\emptyset)$ and $p(\texttt{v3.0.1-alpha}) \rightarrow (3,0,1,\text{alpha})$.

To detect the type of semantic version change, we use a classification function c defined as follows. The function reads the tuples $V_1 = (X_1, Y_1, Z_1, Label_1)$ and $V_2 = (X_2, Y_2, Z_2, Label_2)$ representing two distinct version identifiers. It detects the following version changes:

Major (`X.y.z`): Incremented for incompatible API changes, signaling significant modifications that may require client adjustments.

$$\text{if } X_1 \neq X_2, \text{ then: } \begin{cases} \text{Major Upgrade, if } X_1 < X_2 \\ \text{Major Downgrade, if } X_1 > X_2 \end{cases}$$

Minor (`x.Y.z`): Incremented for adding backward-compatible features, indicating enhancements without breaking existing functionalities.

$$\text{if } X_1 = X_2 \text{ and } Y_1 \neq Y_2, \text{ then: } \begin{cases} \text{Minor Upgrade, if } Y_1 < Y_2 \\ \text{Minor Downgrade, if } Y_1 > Y_2 \end{cases}$$

Patch (`x.y.Z`): Incremented for backward-compatible bug fixes, often associated with routine maintenance updates.

$$\text{if } X_1 = X_2 \text{ and } Y_1 = Y_2 \text{ and } Z_1 \neq Z_2, \text{ then: } \begin{cases} \text{Patch Upgrade, if } Z_1 < Z_2 \\ \text{Patch Downgrade, if } Z_1 > Z_2 \end{cases}$$

Label change (`x.y.z-LABEL`): Updated to reflect the current (e.g., `alpha`, `beta`, `rc`) pre-release stage, indicating the API is not yet ready for production.

$$\text{if } X_1 = X_2 \text{ and } Y_1 = Y_2 \text{ and } Z_1 = Z_2, \text{ then: } \begin{cases} \text{Label Change, if } Label_1 \neq Label_2 \\ \text{No Change, if } Label_1 = Label_2 \end{cases}$$

4.2   API Changes Classification

Each change that occurs affects different elements of a Web API, such as its endpoints, paths, operations, their request/response body, headers, parameters, media types and schemas. The impact of each change depends on how these elements are modified, added, or removed, and whether these changes maintain backward compatibility with existing client implementations.

For example, within the paths, non-breaking changes include the addition of new paths, whereas breaking changes encompass the removal of paths, with further distinctions based on sunset operations and deprecation notices. Changes on operations follow a similar pattern, with the addition of operations being non-breaking, and their removal, especially without sunset dates or before the sunset date, being breaking.

In this paper, we focus on analyzing API changes through specification commit diffs, excluding from them OpenAPI specific modifications that do not impact the API structure, its data model, and security components. We have identified a total of 195 distinct types of changes (See Tables 3 and 4 for some examples). These changes are categorized into three main types: 96 *Breaking Changes*, 66 *Non-Breaking Changes*, and 33 *Undecidable Changes*.

Breaking Changes (**BC**) can disrupt existing client implementations and require clients to adapt to these changes. These changes include modifying existing properties or types (like changing types to enums), adding properties, request parameters or required elements, deleting paths or properties from response payloads, and changing nullable or optional attributes.

Non-Breaking Changes (**NBC**) do not require existing clients to change their implementations. These are generally additive changes such as adding new properties to response payloads, tags, or media types, and changing types where backward compatibility is maintained (like integer to number).

Undecidable Changes (**UC**) refer to those modifications whose impact on the client varies depending on the client's or backend's tolerance level to dealing with unexpected message payloads [3]. For example, when removing authentication or authorization headers, old clients may not break if the security tokens they still send to a tolerant API are ignored. Likewise, properties that are added to API responses may break strict clients which reject unknown data elements. Undecidable changes, cannot be statically classified into breaking or non-breaking without making further assumptions about the client and the API tolerance level. Given that approx. one third of the changes are undecidable, we take them into account with the following two scenarios:

− Best Case Scenario: We assume that all changes classified as undecidable are treated as non-breaking. This perspective allows us to envision a scenario where the potential for disruption due to those changes is minimized.

− Worst Case Scenario: Conversely, the worst-case analysis adopts a more conservative approach by assuming that all undecidable changes have a breaking impact. This stance takes into account a scenario where the ambiguity surrounding those changes is resolved by erring on the side of caution, thereby assuming the maximum possible disruption and compatibility issues.
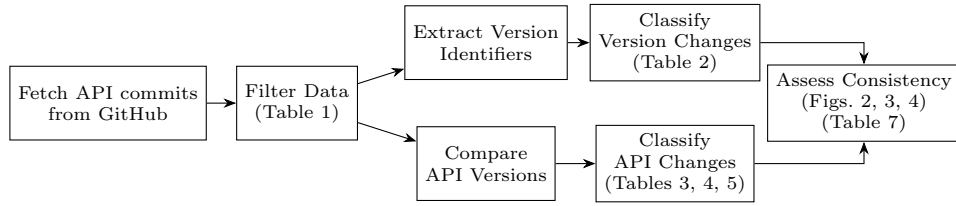
Fig. 1: Data Analytics Pipeline

5   Methodology and Metrics

We implemented a systematic approach to assess consistency between changes detected across API releases (Section 4.2) and the corresponding types of semantic version identifier changes (Section 4.1). The results of the analysis have been obtained by running a pipeline with the following steps (Figure 1).

For each API, we retrieve the complete commits history from its respective GitHub repository. We then ensure that the API meets the filtering criteria as detailed in Table 1. Following, we meticulously sift through the commits to isolate the ones where a version identifier change has happened. The detected version change is then classified to distinguish whether developers have made a Major, Minor, Patch-level release or simply changed the release type label. Following this, we extract the differences between the two consecutive versions of the API, we compare their respective specifications using the `oasdiff` library [11]. The extracted changes are then abstracted by matching them against the known list of 195 change types, which have been pre-classified into the Breaking, Non-Breaking, and Undecidable categories.

The outcome of the pipeline is a table listing, for all APIs and all their releases, the API version change classification with the corresponding API changes. To give a quantitative assessment of the consistency between the two according to semantic versioning rules we compute the following metrics:

– Number of version changes (#VC), further subdivided into the number of Major, Minor, Patch and Label changes (#Major, #Minor, #Patch, #LC)

– Number of API changes (#C), comprising the number of breaking changes (#BC), non-breaking Changes (#NBC), undecidable changes (#UC).

– Proportion of Breaking Changes (BC%):

$$BC\% = \frac{\#BC}{\#C}(\text{Best Case}) \qquad BC\% = \frac{\#BC + \#UC}{\#C}(\text{Worst Case})$$

We assess adherence to semantic versioning by examining if version updates involving at least one breaking change ($\#BC > 0$) or, in the worst-case scenario, at least one undecidable change ($\#UC > 0$), have been accurately categorized as Major. For each API, we define its compliance ratio as $CR = \frac{\#V}{\#VC}$ where $\#V$ is the number of versions which comply with semantic versioning, according to the following rules:

$$\#BC > 0 \implies \text{Major upgrade (Best Case)}$$

$$\#BC > 0 \lor \#UC > 0 \implies \text{Major upgrade} \ \ (\text{Worst Case})$$

This definition permits developers to produce Major releases without introducing breaking changes, as the incompatibility indicated by the version identifier may be due to changes that do not visibly affect the API interface itself.

6   Results

We present the results of the analysis at two levels of granularity. First we quantitatively study each API release independently by characterizing its type of version identifier change and the types of changes introduced in the API itself, by classifying whether they are expected to break or not break clients. This allows us to determine whether the release complies with semantic versioning. Then we proceed to aggregate each release along the history of the corresponding API. This will make it possible to classify the APIs in the dataset according to various facets: which type of changes they underwent at some release in their history, which type of version identifier change, as well as to which extent the API consistently adhered to semantic versioning throughout its entire history. The raw results are publicly shared in a replication package in GitHub.

6.1   Change-level compliance

Types of version changes.   While the most frequently occurring type of version change (Table 2) is the "Patch Upgrade", "Minor Upgrades" can be found more widely across more than half the APIs in the dataset. Overall, the 14 204 Upgrades outnumber the 1 131 Downgrades. As expected, major releases are the least frequent (both concerning upgrades and downgrades). Among the 3 075 APIs, 2 198 APIs have combined at least two types of version changes during their change history. 764 have only one version change. 133 APIs have more than one version change, but they are all of the same type.

Types of API changes In Table 3 we list the most recurrent breaking changes (out of 96). The analysis of breaking change within our dataset prominently highlights "Response property type changed" as the most frequently occurring type of change, followed by the removal of values from enumerated type definitions. The most widespread change affecting 1211 APIs at least once is the removal of paths. Path removal is the complementary change to Path addition, the most prevalent non-breaking change both according to the number of occurrences but also the number (48.14%) of impacted APIs (Table 4).

There is no clear correlation between the presence of specific API changes (e.g., the addition or removal of paths) and the corresponding version identifier changes (listed in the last four columns in the Tables 3 and 4). For example, the removal of paths without deprecation is detected in 246 major releases, which correctly represent the impact of such major change. However, also 629 minor and even 557 patch-level upgrades do include at least one path removal, a clear violation of semantic versioning rules.

Table 2: Classification of version changes (VC) indicating their occurrence (#VC), the total number of breaking, non-breaking and undecidable changes detected in conjunction with each type of version change, as well as their prevalence within all APIs and within how many APIs with breaking changes

	#VC	#UC	#BC	#NBC	#APIs Total	w/BC Best	Worst
Patch Upgrade	7 108	67 032	63 541	77 490	1 669	1 198	1 498
Minor Upgrade	6 038	87 471	54 443	70 820	1 774	1 240	1 412
Major Upgrade	1 058	7 866	11 920	14 854	808	375	422
Label Change	718	3 085	7 938	6 513	345	85	96
Minor Downgrade	459	2 252	4 508	5 160	265	210	233
Patch Downgrade	434	3 056	6 102	3 519	249	210	231
Major Downgrade	238	2 266	2 877	3 560	163	132	150
Total	16 053	173 028	151 329	181 916	3 075	2 148	2 487

Table 3: Most frequent breaking changes

Breaking Change	Occ.	#APIs	#VC	#Major	#Minor	#Patch	#LC
Response Property Type Changed	23 048	714	872	100	335	406	31
Response Property Enum Value Removed	21 210	319	377	43	182	136	16
Path Removed Without Deprecation	15 877	1 211	1 463	246	629	557	31
Response Required Property Removed	12 587	409	547	68	244	205	30
Request Property Enum Value Removed	9 438	223	252	25	114	107	6
Path Parameter Removed	7 019	678	819	118	330	330	41
Response Media Type Removed	5 744	154	168	27	54	77	10
Response Property Pattern Changed	5 032	96	100	6	34	59	1
Response Property Became Optional	4 341	286	333	41	139	135	18
Response Property All Of Removed	4 261	185	240	27	119	87	7
Response Body Type Changed	3 906	351	380	37	170	163	10
Request Property Type Changed	3 872	448	502	40	188	259	15
Response Property Min Length Decreased	3 758	69	73	1	18	51	3
Request Required Property Added	2 524	339	394	62	169	154	9

Table 4: Most frequent non-breaking changes

Non-Breaking Change	Occ.	#APIs	#VC	#Major	#Minor	#Patch	#LC
Path Added	37 928	2 182	2 881	405	1 201	985	290
Response Optional Property Removed	34 172	826	1 011	117	458	413	23
Request Optional Property Added	19 814	1 019	1 259	105	482	627	45
Response Property Became Required	18 112	507	647	80	280	259	28
Request Property Enum Value Added	15 853	324	399	35	178	174	12
Request Optional Parameter Added	12 794	1 343	1 737	447	775	490	25
Response Media Type Added	10 604	334	360	51	148	149	12
Response Non Success Status Added	8 452	704	790	96	362	317	15
Response Optional Header Removed	2 332	73	79	14	52	11	2
Response Property Pattern Added	2 316	81	88	14	31	41	2
Request Parameter Enum Value Added	2 063	148	171	17	77	74	3
Request Parameter Became Optional	1 523	161	165	14	86	65	0
Request Property Became Nullable	1 493	111	135	8	76	39	12
Request Property Became Optional	1 433	257	293	36	131	112	14
Request Optional Default Parameter Added	1 122	69	75	7	29	38	1
Response Success Status Added	1 052	315	336	53	129	151	3
Response Required Property Became Not Read-Only	925	15	21	0	9	8	4

Table 5: All the non-breaking changes that were associated with a Major version change during which no breaking changes occurred

Non-Breaking Change	Occurrences	#APIs	#VC(=#Major)
Request Optional Parameter Added	917	333	334
Path Added	763	118	141
Response Non Success Status Added	214	20	21
Response Optional Property Removed	10	5	5
Response Success Status Added	6	3	4
Request Parameter Became Optional	6	3	3
Request Optional Default Parameter Added To Existing Path	5	1	1
Response Media Type Added	3	3	3
Request Optional Property Added	2	1	1
Request Property Became Optional	2	1	1
Request Property Enum Value Added	2	1	1
Request Parameter Enum Value Added	1	1	1

Version Changes classification by API Change Type. How many major releases contain at least some breaking changes? According to the aggregated results in Table 2 – listing the total number of breaking, non-breaking and undecidable changes for each type of version change – there are 1 058 major upgrades with 7 866 breaking changes in total. While according to semantic versioning, there should be no breaking changes for patch and minor upgrades, we can read that the highest number of breaking changes (87 471) is actually detected in conjunction with minor upgrades. Notably, label changes, despite their lower frequency, also account for a significant number of breaking changes, indicating that clients can and will be broken as an API `alpha` release is updated to `beta`.

The total number of breaking changes listed in Table 2 is further decomposed in Table 6 with some statistics. It stands out that the worst major release introduced 723 breaking changes. This is a small number, however, if compared to the 2 508 breaking changes applied to one *minor* release. We also spot that the minimum number of breaking changes is 0 across all version change types. This means that there at least some minor releases without breaking changes. How many? Only 32% of the minor releases and 27% of the Patch releases do include exclusively non-breaking changes as we can see from Fig 2, showing a complete, detailed map of the major, minor and patch version changes classified according to the corresponding mix of API change types. For example, we can see that while 705 major releases of 375 APIs contain at least one breaking change, 75 releases contain *only* breaking changes. In the worst case, 813 major releases of 422 APIs contain both at least one breaking and one undecidable change. There, we also observe that 37% of major releases include only non-breaking changes, all of which are listed in Table 5.

Non breaking changes in Major releases. While it is not a violation of semantic versioning to launch a major release that is fully backwards compatible, we observed that there is only a limited number of 12 non-breaking changes when this happens (Table 5). Predominantly, the most frequent changes pertained to

Table 6: Number of breaking changes detected for each type of version change

| #BC (Best)	Max		Min		Average		Median		StdDev	
#BC+#UC (Worst)	Worst	Best	Worst	Best	Worst	Best	Worst	Best	Worst	Best
Major Upgrade	723	509	0	0	18.70	11.27	1	0	64.97	42.77
Minor Upgrade	2 508	2 508	0	0	23.50	9.02	2	0	101.37	57.34
Patch Upgrade	2 308	1 692	0	0	18.37	8.94	2	0	94.25	57.29
Major Downgrade	553	518	0	0	21.61	12.09	5	3	50.60	39.11
Minor Downgrade	362	246	0	0	14.73	9.82	4	3	32.27	20.14
Patch Downgrade	559	349	0	0	21.10	14.06	4	3	58.60	43.12
Label Change	2 637	2 596	0	0	15.35	11.06	0	0	110.29	105.17
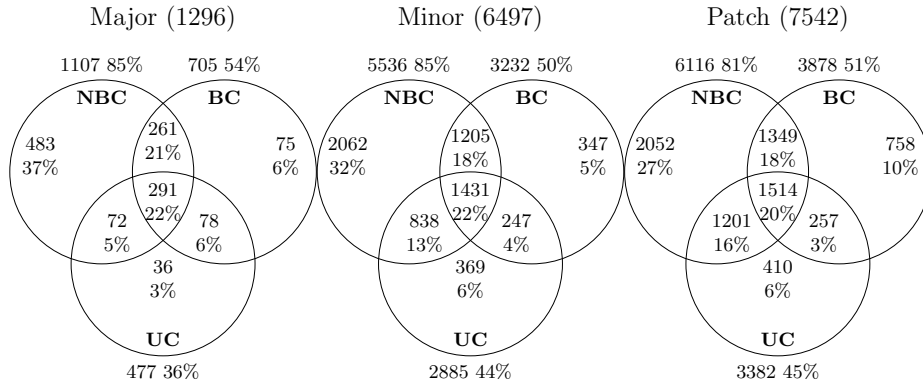
Fig. 2: Classification of the Major, Minor and Patch-level releases according to their mix of breaking (BC), non-breaking (NBC), and undecidable (UC) changes. The values outside the circles refer to the number of version changes with at least one type of API change

modifications in the API structure, such as the inclusion of new paths or the addition of optional request parameters.

Version Change vs. Breaking Change Proportion. While in 594 major, 2 778 minor, and 3 220 patch releases do include changes of exactly one type, 54% of major releases (57% of minor and also 57% of patch) do include a mix of changes. It is thus worth to investigate how the proportion of breaking changes (BC%) relative to all changes influences the decision for a version upgrade. Figure 3 illustrates the BC% distribution both for the best and worst cases, with the APIs segmented according to the type of version change involved (Major, Minor, Patch, Label Change) as well as whether the version was upgraded (top) or downgraded (bottom). The 'Normalized Frequency' plots within the main histograms provide a relative comparison, allowing for the visual assessment of the impact of the proportion of breaking changes on the decision to launch a major or minor release irrespective of the absolute number of version changes.

In both the best and worst-case scenarios, the histograms show that most version changes have a null proportion of breaking changes, as evidenced by
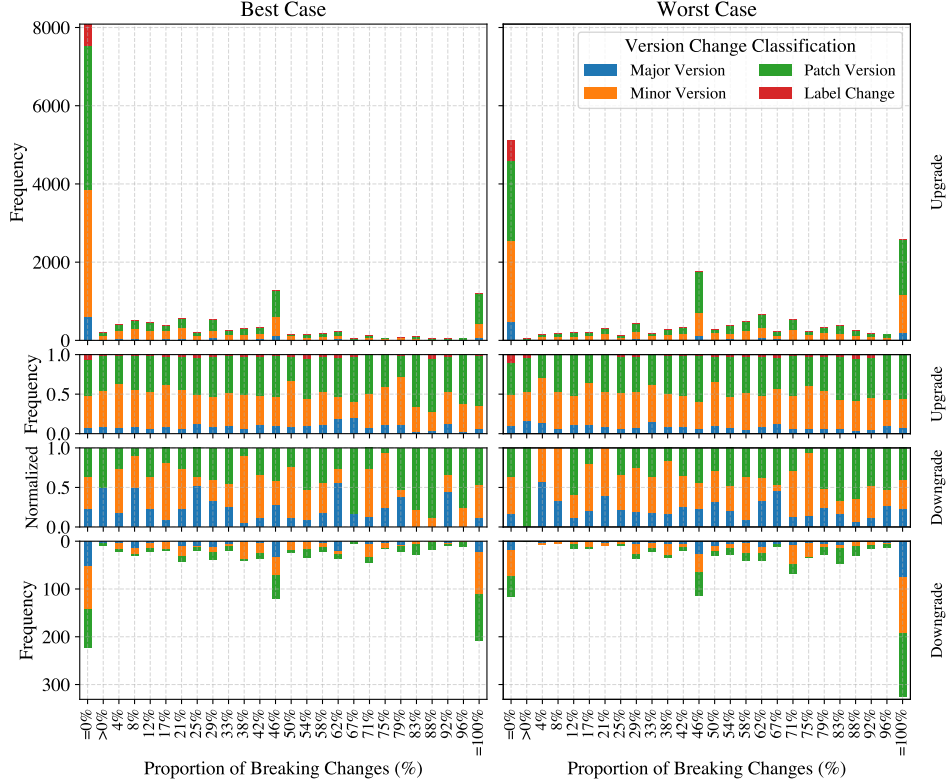
Fig. 3: Breaking changes proportion distributions for Upgrades (above) and Downgrades (below), categorized by each type of version change

the high bars at the left side of the histograms (BC% = 0%). This observation is consistent with the fact that 54.68% of the APIs exclusively undergo non-breaking changes, thus maintaining backward compatibility. The presence of bars across all intervals indicates that breaking changes are spread across the entire spectrum, becoming more and more prevalent, up to thousands of releases which include only breaking changes. The normalized plots reveal that, regardless of whether updates are classified as upgrades or downgrades, the proportion of breaking changes does not significantly affect the assignment of a new version number to the API. This trend persists even in cases where breaking changes constitute 100% of the alterations, indicating scenarios where all the changes were breaking and developers still assigned a non-major version to the release. In the worst-case scenario, we identified that there were 66 distinct types of breaking changes that were applied in the absence of any non-breaking ones.
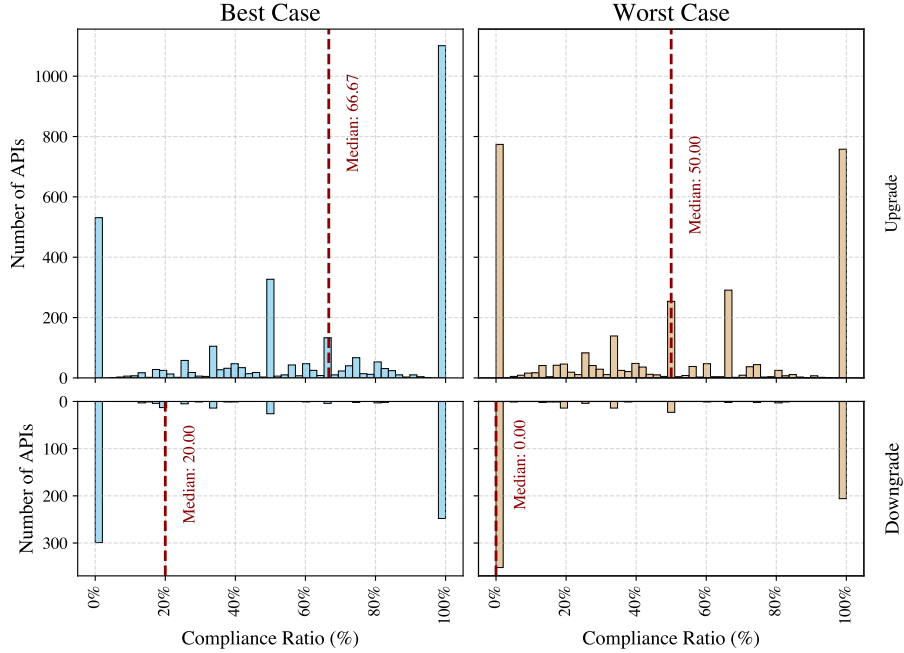
Fig. 4: Compliance Ratio Distribution

6.2   API-level Compliance

APIs that adhere to semantic versioning are those that have consistently maintained backward compatibility or have appropriately notified clients of any compatibility breaks through version identifiers. Within our dataset, under the best case scenario, we identified a total of 962 adhering APIs (out of 3075 that experienced at least one instance of breaking changes (BC), non-breaking changes (NBC), or undecidable changes (UC)). In the worst-case scenario, this number decreases to 588 APIs. When examining the subset of 2487 APIs that introduced breaking changes, we found that 517 APIs in the best case and only 180 in the worst case have adhered to semantic versioning principles (Table 7). These APIs have the highest average number of major releases. The highest average number of releases (#VC) overall, however, is found within the non-compliant APIs. These also underwent a significantly larger number of changes (484 221) than the APIs which adhere to semantic versioning (31 244).

Figure 4 provides a nuanced view of the compliance ratio for both best and worst-case scenarios also distinguishing upgrades from downgrades. It illustrates that only some APIs do consistently adhere to (1 444 in the best case, 768 in the worst) or always deviate (532 in the best case, 766 in the worst) from compliance across all releases. Instead, there is a non-empty subset of 1541 APIs with partial compliance in the worst case. The central peak with 50% compliance ratio accounts for the 582 APIs with two releases, out of which only one is compliant.

Table 7: Metrics comparison for APIs classified according to their compliance

	Adhering to Semantic Versioning				Not Adhering		Total
	BC%= 0		BC%> 0		BC%> 0		
Metric	Best	Worst	Best	Worst	Best	Worst	
#APIs	927	588	517	180	1 970	2 307	3 075
#VC	2 190	1 089	1 527	413	14 423	15 537	16 053
Avg #VC	2.36	1.85	2.95	2.29	7.32	6.73	5.22
Avg #Major	0.17	0.32	0.24	0.86	0.04	0.04	0.08
Avg #Minor	0.27	0.32	0.20	0.10	0.45	0.44	0.42
Avg #Patch	0.33	0.26	0.29	0.04	0.49	0.48	0.46
#BC	0	0	2 723	2 665	148 606	148 664	151 329
#NBC	8 226	4 558	7 584	3 896	169 774	173 462	181 916
#UC	5 523	0	7 188	1 440	165 840	171 588	173 028
Avg BC% (Best)	0.00	0.00	8.81	31.46	24.75	23.00	11.19
Avg BC% (Worst)	12.73	0.00	30.20	42.37	46.38	44.90	29.77

7　Discussion

How often APIs introduce breaking vs. non-breaking changes?

The analysis of histories of 3,075 APIs that experienced changes affecting their functionalities, revealed that 80.87%, included backward incompatible changes. This finding reveals the considerable challenge developers face in maintaining backward compatibility. The prevalence of such changes underlines the critical need for effective versioning strategies and comprehensive documentation to mitigate potential disruptions and ensure a smoother transition for API consumers.

Are there many Web APIs which consistently follow semantic versioning rules across their entire history?

Contrary to theoretical expectations, the study uncovered that only 577 APIs with breaking or potentially breaking changes adequately reflected these alterations by launching a major release, adhering to semantic versioning principles in practice. Moreover, despite SemVer guidelines suggesting that minor versions should only introduce backward-compatible features, 2 282 APIs did release breaking changes as minor or even patch-level updates. This deviation could be due to a misinterpretation of what constitutes a breaking change or a desire to push new features quickly without incrementing the major version.

We also found 910 APIs where Major version updates did not introduce any breaking changes. Interestingly, these non-breaking changes (NBC) were categorized into exactly 12 distinct types. This observation suggests a nuanced approach to versioning, where developers might choose to launch major releases for reasons other than breaking changes, such as significant feature additions or improvements meant to attract new clients without breaking existing ones.

7.1   Threats to validity

Construct Validity.   Not all changes may be documented or detected, especially if they are subtle or indirect, possibly underestimating the true impact of API changes [21]. We rely on a single tool (`oasdiff`) to detect the changes, which may bias the results. The classification of the changes into breaking, non-breaking, and undecidable was manually performed by the authors.
External Validity.   A potential threat to the generalizability of the findings arises from the focus on Web APIs specifications hosted on GitHub, raising questions about the applicability of our findings to proprietary or non-GitHub hosted API documentation.
Internal Validity.   Establishing a clear causal relationship between the presence of specific types of API changes and the corresponding type of version identifier change remains an open challenge. The decision on which type of major, minor or patch-level release may be influenced by other confounding factors. As most APIs have a only a few releases in their histories, this may produce discretization artifacts in the distributions shown in Figures 3 and 4.

8   Conclusion

The results of the study presented in this paper underscore a critical need for tools and guidelines tailored specifically for correctly applying semantic versioning to Web APIs. With an empirical analysis tracking the evolution histories of 3 075 Web APIs, we found that in the worst case (assuming clients and backends perform strict checking of message payloads) only 768 (25%) APIs consistently comply with Semantic Versioning by always releasing major upgrades for breaking changes (180), or never breaking their backward compatibility (588). This number grows to 1444 APIs (46%) when assuming clients and backends follow the "tolerant reader" pattern [3].

   This finding highlights a discrepancy between the theory [1] and the state of the practice of semantic versioning within the Web APIs described using OpenAPI specifications, tracked using GitHub open source repositories. Based on these results, there is a need for establishing standardized versioning protocols which can be embedded into semantic versioning calculators [7,22] to mitigate the observed inconsistencies, benefiting both Web API developers and consumers by enhancing predictability, reducing potential disruptions, simplifying dependency management, and fostering a more resilient Web API ecosystem.

References

1. Semantic Versioning. `https://semver.org/`
2. Bogart, C., Kästner, C., Herbsleb, J., Thung, F.: When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. ACM TOSEM **30**(4), 1–56 (2021)
3. Daigneau, R.: Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services. Addison-Wesley (2012)

4. 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)
5. Giretti, A.: API versioning. In: Beginning gRPC with ASP.NET Core 6. pp. 223–237 (2022)
6. Henning, M.: API design matters. Queue **5**(4), 24–36 (2007)
7. Lam, P., Dietrich, J., Pearce, D.J.: Putting the semantics into semantic versioning. In: Proc. of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. p. 157–179 (2020)
8. Lamothe, M., Guéhéneuc, Y.G., Shang, W.: A systematic review of api evolution literature. ACM Computing Surveys (CSUR) **54**(8), 1–36 (2021)
9. Li, W., Wu, F., Fu, C., Zhou, F.: A large-scale empirical study on semantic versioning in golang ecosystem. In: Proc. 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1604–1614. IEEE (2023)
10. Medjaoui, M., Wilde, E., Mitra, R., Amundsen, M.: Continuous API management. O'Reilly (2021)
11. OASDiff: `https://github.com/Tufin/oasdiff`
12. Ochoa, L., Degueule, T., Falleri, J.R.: Breakbot: Analyzing the impact of breaking changes to assist library evolution. In: Proc. 44th International Conference on Software Engineering (ICSE): New Ideas and Emerging Results. pp. 26–30 (2022)
13. 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)
14. OpenAPI Initiative: `https://www.openapis.org/`
15. Peralta, J.H.: Microservice APIs: Using Python, Flask, FastAPI, OpenAPI and More. Simon and Schuster (2023)
16. Pinckney, D., Cassano, F., Guha, A., Bell, J.: A large scale analysis of semantic versioning in npm. In: IEEE International Working Conference on Mining Software Repositories (2023)
17. Raemaekers, S., van Deursen, A., Visser, J.: Semantic versioning and impact of breaking changes in the maven repository. Journal of Systems and Software **129**, 140–158 (2017)
18. Serbout, S., Muñoz Hurtado, D.C., Pautasso, C.: Interactively exploring api changes and versioning consistency. In: Proc. 11th IEEE Working Conference on Software Visualization (VISSOFT). pp. 28–39 (October 2023)
19. Serbout, S., Pautasso, C.: An empirical study of web api versioning practices. In: International Conference on Web Engineering. pp. 303–318. Springer (2023)
20. Serbout, S., Pautasso, C.: APIstic: A large collection of OpenAPI metrics. In: Proc. 21st IEEE/ACM International Conference on Mining Software Repositories (MSR). Lisbon, Portugal (April 2024)
21. Stocker, M., Zimmermann, O.: API refactoring to patterns: Catalog, template and tools for remote interface evolution. In: Proc. 28th European Conference on Pattern Languages of Programs (EuroPLoP). ACM (2023)
22. 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. of the 37th IEEE/ACM International Conference on Automated Software Engineering. pp. 1–12 (2022)
23. Zimmermann, O., Stocker, M., Lubke, D., Zdun, U., Pautasso, C.: Patterns for API design: simplifying integration with loosely coupled message exchanges. Addison-Wesley (2022)