Interactively Exploring
API Changes and Versioning Consistency

Souhaila Serbout [iD]
Software Institute
USI Lugano, Switzerland
souhaila.serbout@usi.ch

Diana Carolina Muñoz Hurtado [iD]
Software Institute
USI Lugano, Switzerland
carolina.munoz@usi.ch

Cesare Pautasso [iD]
Software Institute
USI Lugano, Switzerland
c.pautasso@ieee.org

Abstract—**Application Programming Interfaces (APIs) evolve over time. As they change, they are expected to be versioned based on how changes might affect their clients. In this paper, we present two novel visualizations specifically designed to represent all structural changes and the level of adherence to semantic versioning practices over time. They can also serve for characterizing and comparing the evolution history of different Web APIs. The API VERSION CLOCK helps to visualize the sequence of API changes over time and highlight inconsistencies between major, minor, or patch version changes and the corresponding introduced breaking or non-breaking changes applied to the API. The API CHANGES overview aggregates all changes to an OpenAPI (OAS) description, highlighting the unstable vs. the stable elements of the API over its entire history. Both visualizations can be automatically created using the APICTURE, a command-line and web-based tool that analyzes the histories of git code repositories containing OAS descriptions, extracting the necessary data for generating visualizations and computing metrics related to API evolution and versioning. The visualizations have been successfully applied to classify, compare, and interactively explore the multi-year evolution history of APIs with up to hundreds of individual commits.**

Video URL: https://youtu.be/WtFm6VvKi20

Index Terms—**Web API Evolution, Semantic Versioning, Change Visualization, Sunburst Chart**

I. INTRODUCTION

The evolution of Application Programming Interfaces (APIs) is a dynamic process that requires careful monitoring and analysis [17, 27]. As APIs undergo continuous updates [20], it becomes essential to understand [9] and control [32] the changes that occur over time to ensure seamless integration [51], maintain backward compatibility [24], track the progress of development efforts [14], and assess how their quality and performance evolves over time [8, 15].

In this paper, we use the sunburst visualization towards understanding the evolution of Web APIs, specifically concerning tracking the co-evolution of API structures and their versioning metadata. The main goal is to characterize and compare how different Web APIs evolve over large periods of time, in order to visually identify different API evolution patterns [22, 31, 46]. We introduce visualizations designed to distinguish which API elements have changed often, how such changes impact clients [18, 29, 49], and whether API

developers consistently update versioning metadata to control client expectations about the impact of such changes [25]. The interactive visualizations and their scalability have been tested by using it to explore a large collection of 3,271 API change histories mined from open source GitHub repositories. In this paper we include a small sample, showing a rich diversity of Web API evolution histories reflected in our visualizations.

More in detail, the API VERSION CLOCK visualization shows when each API change happened, what is their impact on clients (e.g., classifying breaking vs. non-breaking changes), and their relationship with the API versioning metadata. The goal is to help API developers reflect on the pace of their API evolution and remind them to bump the versioning metadata to consistently reflect the impact of changes on their API clients. The API CHANGES visualization complements it by precisely representing which elements of an OpenAPI (OAS) description [5] have changed. The visualization can be applied both to study individual diffs, but also cumulatively over sets of changes up to the entire API history. The visualization highlights the unstable elements of an API description by showing how they have changed during a defined timeframe.

The visualization is supported by APICTURE, a CLI tool that offers the convenience of generating visualizations directly from git repositories that already contain an OAS specification. The tool can be embedded into DevOps build pipelines [11] to generate updated visualizations at every commit so that API developers can effortlessly track and understand the evolution of their Web APIs.

This paper makes the following contributions:

1) the API VERSION CLOCK visualization classifies all changes based on their impact on clients and captures them in chronological order while putting them in relationships with the different API versions.

2) the API CHANGES visualization provides a precise localization and analysis of modifications within the API description structure. This visualization aids in understanding the stability of specific API elements and fine-grained evolution patterns of the API design.

3) the integration of API CHANGES with API VERSION CLOCK to provide a holistic view of the API's evolution journey, supporting the analysis of versioning practices and
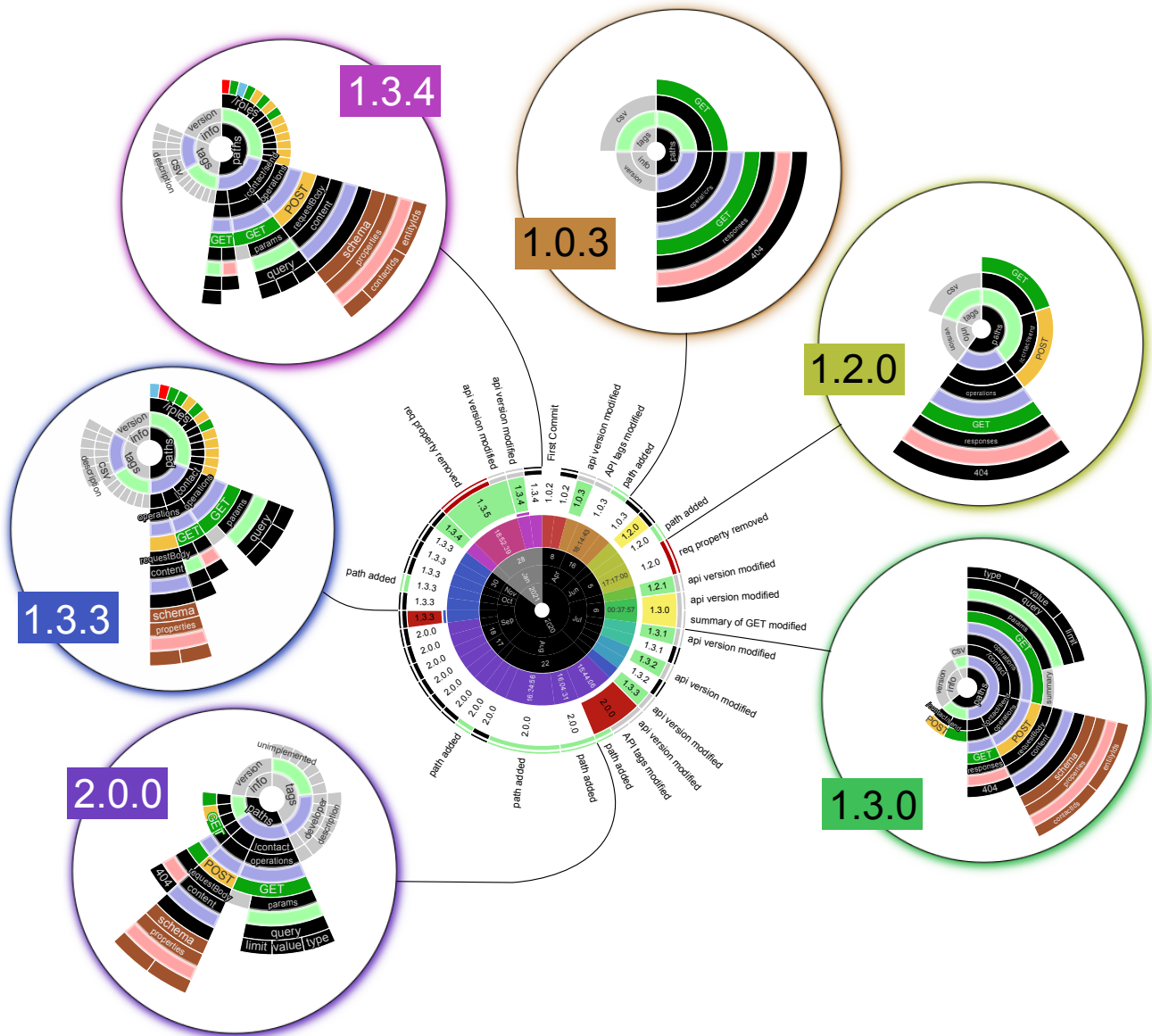
1

Fig. 1. API VERSION CLOCK (center) and API CHANGES until a given version of the Bmore Responsive API. Legend explained in Figures 2 and 4.

the localization of breaking and non-breaking changes on the API structure and data model.

4) a small gallery of API evolution visualizations, derived from real-world APIs that test the scalability of the visualization to increasingly larger histories (both in terms of versions and commits) and exhibit a variety of API evolution patterns.

5) the APICTURE tool, which automates the process of generating both visualizations, providing API developers and stakeholders with an interactive means to analyze, visualize and reflect on their API evolution and versioning strategies.

The rest of this paper is structured as follows: In Section II we present an API example which we use to demonstrate the visualization use cases. In Sections III and IV we present API VERSION CLOCK and API CHANGES. In Section V we include a gallery of selected real-world API evolution

examples and explain insights gained from their visualizations, then we discuss in Section VI the limitations of the proposed visualizations and areas or improvements. Finally in Sections VII, VIII and IX we present related work, draw some conclusions and outline our future research agenda.

II. USE CASE SCENARIOS AND EXAMPLE API

In this section we use a real-world API to introduce and explain the design of the two visualizations and how they are generated by leveraging the API's git historical record of changes. The Bmore Responsive API [2] is an emergency response and contact management API designed for monitoring and coordinating emergency responses to critical scenarios. The API history includes a total of 49 commits spanning from the first commit on *April 6, 2020* to the last commit on *February 28, 2022*. Throughout its 693-day-long history,

we found 13 distinct versions of the API (from `1.0.2` to `2.0.0`, later reverted back to `1.3.4`).

Developers are interested in reflecting on the history of the API to answer the following questions [14, 17, 32]:

- Q1: Did we correctly and consistently follow a semantic versioning [6] strategy?
- Q2: How often did we revert the API to a previous version?
- Q3: Did we follow a regular API maintenance and update cycle over time?
- Q4: Did we always ensure backwards compatibility of the introduced changes?
- Q5: Which are the stable and the unstable parts of the API structure and data model?
- Q6: Can we detect if our API followed a unique evolution path compared to other ones?

The interactive visualizations introduced in this paper are intended to help both API developers and developers of API clients to answer such questions.

In particular, the API VERSION CLOCK visualization uses the sunburst plot to provide a compact chronological view of the flow of changes over time. With it, developers can observe the progression of API versions. On the other hand, the API CHANGES visualization uses the sunburst hierarchy to offer a detailed representation of the accumulated changes that have occurred within a specific time frame. This visualization aids in localizing the specific areas of the API structure where the changes have occurred. Figure 1 exemplifies this integration, demonstrating how and when changes flow through specific time points in the history of the Bmore Responsive API. With the assistance of the API CHANGES visualization, it becomes apparent which parts of the API have been affected by these changes, enabling a more granular analysis of the API's evolution.

Both visualizations represent the evolution of an API from a different and complementary perspective. By default they display the entire API history, aggregating all changes into a single plot. If API designers are interested to observe which change happened when they can use the API VERSION CLOCK to select a specific commit so that the corresponding API CHANGES can show what API elements changed since the previous commit. Likewise, they can select to view all changes leading up to a certain version of the API, or all changes that happened between two different releases. As mentioned above, each API modification represented in API CHANGES can be contextualized along the time dimension thanks to the API VERSION CLOCK, e.g., by highlighting the commit in which they occur.

III. API VERSION CLOCK VISUALIZATION

A. *Visualization goal*

This visualization serves as an evolutionary clock, providing a visual representation of the different types of changes that occur during a specific timeframe in an API's history. As depicted in Figure 2, in the *Time* ring – the fourth one from
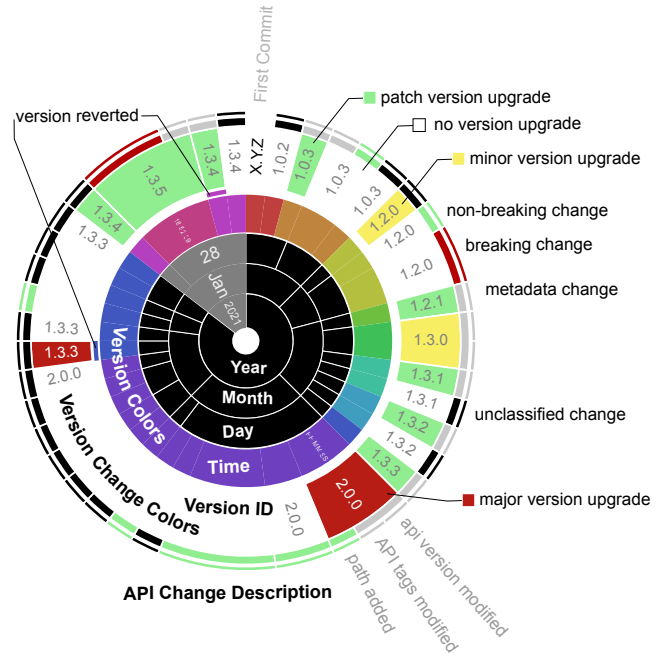

Fig. 2. API VERSION CLOCK Design visualizing the Bmore Responsive API

the center – each version upgrade is assigned a unique color, allowing for a clear depiction of the progression of version identifiers and the corresponding types of changes – shown in the outer rings – throughout the entire history of the API. Notably, for APIs that adopt the semantic versioning format, the visualization incorporates, in the *Version ID* ring, color coding to differentiate between major, minor, and patch-level upgrades.

The primary objective of API VERSION CLOCK is to assess the congruence between version identifier updates and the relative significance of breaking and non-breaking changes introduced in the API over time. Developers can leverage this visualization to gain insights into the adopted versioning strategy for a specific API and evaluate its adherence to the principles of semantic versioning.

In the case of the Bmore Responsive API, according to the API VERSION CLOCK visualization in Figure 2, semantic versioning is not properly adopted. Breaking changes were introduced however this was not reflected by the version identifiers. Moreover, the version identifier has been reverted twice during the API's history.

B. *Building* API VERSION CLOCK

As illustrated in Figure 3, the construction of the visualization involves retrieving all the git commits that made modifications to the OAS description of the API. We then compute the differences between each pair of consecutive commits, allowing us to identify the specific changes made during each commit. For extracting the changes we rely on *oasdiff* [4], an open source command-line tool and Go package that compares two OAS descriptions.
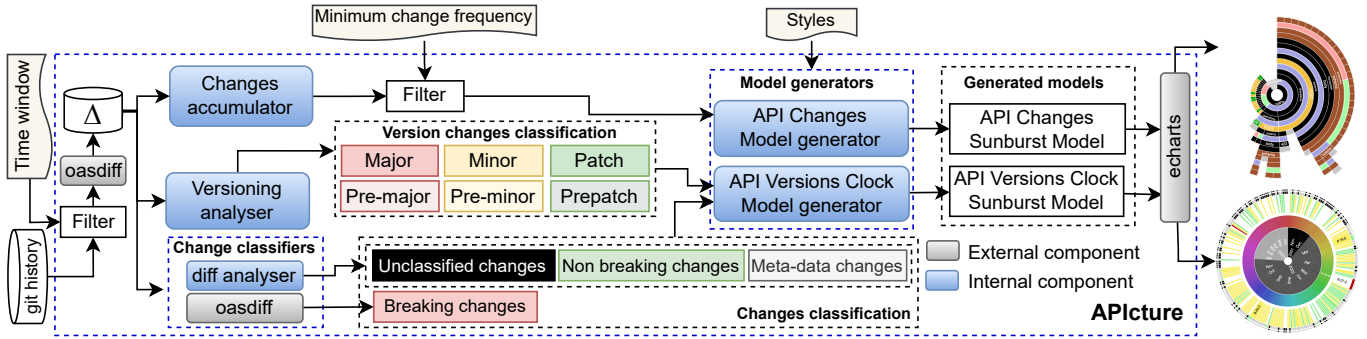
Fig. 3. API VERSION CLOCK and API CHANGES analysis and rendering pipeline

The extracted differential data serves as input for both the *Versioning analyzer* and the *Change classifiers*, which generate classification results regarding version changes and the types of changes (breaking, non-breaking, metadata, unclassified). These outputs, along with the necessary visual elements described in Figure 2, are utilized by the *API Version Clock model generator*, which then constructs the sunburst model. Finally, the generated model is rendered using ECharts [28].

The *Change classifiers* consists of two components: *oasdiff* [4], the external tool responsible for detecting breaking changes, and *diff analyzer* a tool developed as part of our work, which helps classify changes as non-breaking or unclassified.

Breaking changes are modifications made to an API that disrupt existing functionalities and result in backward compatibility issues with the latest deployed version [50]. These changes are identified and extracted from the differential data using again *oasdiff*. Conversely, non-breaking changes refer to modifications that do not introduce incompatibilities with existing functionality or the ability of clients to interact with the API. Unclassified changes, in the current version of the tool, are non-breaking changes which we could not precisely determine which parts of the API they affect.

C. Visualization Structure

In Figure 2, we illustrate the structure of the obtained API VERSION CLOCK visualization:

• *Localizing change in its temporal context:* The commits timestamps are mapped to the core rings of the sunburst visualization, starting from the year, month, day, and time of each commit. These rings provide a temporal context for the visualization, allowing users to observe the chronological order of the commits in a clockwise direction. They also allow users to hierarchically filter along the time axis by selecting to visualize only commits of a specific year, month, or day.

• *Localizing version change in its temporal context:* Since the evolutionary analysis of our study relies on the git commit history, the fourth ring of the version change visualization plays a crucial role in indicating the exact timestamp of each commit. This ring is color-coded based on the API version associated with that particular timestamp, allowing for a clear visual representation of version changes over time (Figure 2). A unique color is associated with each version identifier,

currently based on a simple uniform mapping to the HUE component of HSL color values.

• *Discerning types of version change:* While the inner core of the visualization is dedicated to the time of each commit, the next layer represents the version identifier of the API at that time, extracted from the standard OAS metadata. While the fourth ring highlights the version change, in the case where semantic versioning is used, we distinguish in the *Version ID* ring the types of version change by assigning a specific color for each of the ■ major, ▢ minor, and ▢ patch releases.

With our analysis on version identifier changes [43], we could detect the presence of commits where the version was reverted to a previous identifier. To highlight such backward evolution steps. An example of this case is happening in Bmore responsive API, where the version was reverted from 2.0.0 back to 1.3.0 again in one of the commits in late September 2020. As can be seen from Figure 2, we added an extra thin ring sandwiched between the colored API version ring and the following one. The color of the ring is the same as the version to which the revert happened. This ring will remain empty for APIs with a monotonic version identifier evolution and highlight backward versioning steps otherwise.

• *Detecting API backward incompatibility:* Within the context of API elements, changes primarily pertain to modifications in the structure of endpoints, request/response schemas, and security/authentication mechanisms. These alterations directly impact the functional aspects of the API, potentially introducing *breaking changes* or enhancements to its capabilities. On the other hand, changes in the metadata predominantly involve updates to descriptive attributes that provide contextual information about the API. These include modifications to the API title, version number, server URL, and details about the API provider. Such changes are typically non-breaking in nature and focus on improving the clarity, documentation, or administrative aspects of the API [21]. Within each version segment, the sunburst chart displays on the outer rings, how many changes were detected by comparing the current commit against the previous commit. These changes are depicted first by distinguishing the breaking changes from the non-breaking ones and then – in the outer ring – by further indicating the presence and the amount of specific types of changes.

• *Assessing Semantic Versioning Compliance:* To facili-

tate the assessment of compliance with semantic versioning practices, we intentionally utilize the same color scheme to represent both version identifier changes and the classification of changes as breaking or non-breaking. This design choice allows for easy visual identification of whether the API evolution aligns with correct semantic versioning principles. Commit nodes associated with breaking changes should correspond to major version upgrades (■), while non-breaking changes should be observed alongside patch-level version upgrades (□). By examining the placement and distribution of these node types, we expect users to readily detect instances where changes are not in accordance with the expected versioning rules.

D. API VERSION CLOCK *Interactive Features*

The visualization employs interactive ECharts features for dynamic exploration of the sunburst plot structure at various levels through *zoom in and out*. The *zoom* option aids in examining changes within specific timestamps or periods. Tooltips are essential, showing the change count for a slice. For time rings, the tooltip displays how many changes occurred at a given time.

IV. API CHANGES VISUALIZATION

A. *Visualization goal*

The purpose of this visualization is to provide a clear understanding of the location of the changes occurring in the API structure, its data model as well as related metadata information. By visually representing the ▨ deletions, ▨ additions, and ▨ modifications happening at the level of ▨ metadata, ■ structural and ▨ data model elements, it becomes easier to *localize* the frequently occurring changes and comprehend the overall stability of the API design (Figure 4).

One important aspect to highlight is that the changes visualization does not indicate when each change occurred. This deliberate omission contributes to the scalability of the visualization, as it allows for the accumulated changes across multiple API versions and commits within a specific timeframe to be captured in one visualization. This approach simplifies the visualization process and enables a more efficient analysis, particularly when dealing with APIs with long histories or many fine-grained granular changes applied to it.

B. *Building* API CHANGES

To build the visualization, we use the same differentials extracted for API VERSION CLOCK (See Figure 3) and transform them into a sunburst model so that it can be rendered using ECharts [28]. The structure of the sunburst tree reflects the OpenAPI specification structure. However, we only include OpenAPI elements which change at least once. As we accumulate all changes from multiple commits, we keep track of many times each element change, which is reflected in the relative ring sector angle. The elements represented in the sunburst plot are colored as defined in Figure 4.

In the API CHANGES visualization, instead of explicitly displaying individual changes like in the API VERSION
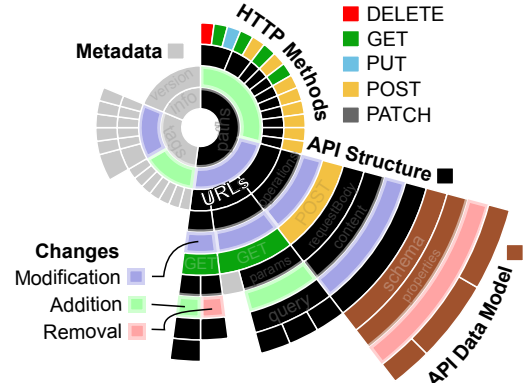

Fig. 4. API CHANGES Design applied to the BMore Responsive API

CLOCK visualization, we employ an abstraction that focuses on measuring the frequency of different types of changes.

APICTURE provides users with CLI options that offers the ability to customize the API history timeframe of interest. Specifically, in the context of API CHANGES visualization, APIcture allows users to filter out changes occurring less frequently than a defined minimum threshold. These are represented as *Filters* in Figure 3. Note that the *Time window Filter* can also be used to customize the history time frame for API VERSION CLOCK.

C. *Visualization structure*

We tailored the sunburst visualization to characterize the nature and also represent the magnitude of the changes occurring throughout the API lifespan and identify the unstable API elements that have undergone more frequent modifications.

• *API changes localization:* The nature of the changes shown in the visualization is contingent upon whether they occur within the *API structure elements*, *API datamodel* or the *metadata* elements, encompassing description fields, API title, API version, server URL, and API provider information. By distinguishing between changes in each of the later elements, the visualization enables a more comprehensive understanding of the different facets of changes occurring within the API evolution. This distinction allows developers and stakeholders to assess the amount of changes impacting both the functional behavior of the API and its associated contextual information.

By presenting the changes in a way mimicking a dereferenced version of the original OAS specification tree structure, it becomes easier to discern the exact areas or elements of an API that have undergone most alterations over time. The goal is to draw the attention the API elements affected by changes during its history. The wider the ring sector angular extent, the more frequently the corresponding API element changes. Different change actions affecting the API elements are highlighted using distinct colors (Figure 4): ▨ Deleted, ▨ Added, ▨ Modified. These colored sectors refer to the type of changes applied to the element representing in their parent sector, the one found immediately above towards the center
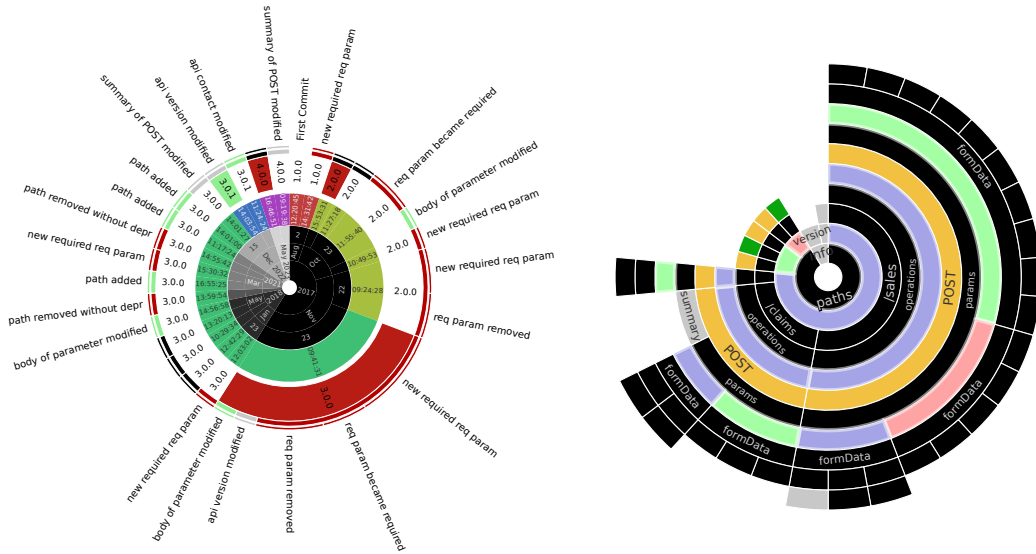
Fig. 5. Visualizations of the SunRocks API Evolution (24 commits over 5 versions during 2114 days, 0★)

of the sunburst. These light-colored action sectors serve as a visual marker, drawing attention to the specific location within the API structure where the modification has taken place.

However, it should be noted that for non-object elements, such as the metadata element, and certain data types (e.g., enumeration), the only discernible alteration that can be detected is when the value itself changes. In these cases, there are no deeper levels or nested API elements to highlight, as the modification is confined to the value itself.

• *Quantifying change granularity:* Change granularity can be assessed by examining the *level* (depth of the *rings*) associated with a specific API element. For instance, if a specific endpoint is added (i.e., addition for a path), it represents a *less fine-grained change* as compared to the addition of a specific query parameter. This can be seen in API CHANGES of Bmore Responsive API in Figures 1 and 4. Similarly, the addition of a new property to the schema of a response object for a specific operation signifies a *more fine-grained change* when contrasted with the addition of a parameter.

D. API CHANGES *Interactive Features*

Like the API VERSION CLOCK, the API CHANGES visualization utilizes ECharts' interactive features. Here, zooming hones in on changes within API elements. Clicking an element expands rings for detailed insight into nested elements and changes. Tooltips show element frequency and labels. Labels are hidden when angles are tight, enhancing readability. APICTURE allows customization by time frame and minimum frequency for tailored API CHANGES visualization.

V. API EVOLUTION GALLERY

In this section, we present five API evolution examples (Figures 5–9) growing from 24 commits up to 144 commits. They were selected out of a dataset of 3271 APIs, as they present different characteristics in terms of their evolution

dynamics, their use of semantic versioning, the reached level of maturity, and their co-evolution with different repository artifacts. In the captions, we report the size of their evolution history (number of commits, versions, and duration in days) and the number of GitHub stars for the corresponding repository, which were also considered during the example selection process. The visualization gallery is obtained from a snapshot of the corresponding git repositories taken on 26 June 2023.

A. SunRocks API Evolution

The SunRocks's API VERSION CLOCK visualization provides a chronological depiction of the changes occurring in the SunRocks API from version `1.0.0` to version `4.0.0`, spanning over a period of more than 5 years (Figure 5 left). The majority of breaking changes took place within the first year, coinciding with two major version upgrades. No minor version increments were observed, and only a single patch version upgrade occurred (`3.0.1`), accompanied by a metadata change.

In all major version upgrades, there were always some unclassified or breaking changes, indicating that each upgrade involved modifications with a potential impact on the API backward compatibility. Upon closer examination through the API CHANGES Visualization (Figure 5 right), it becomes evident that the changes primarily manifested at the parameter level of POST methods of the `/claims` and `/sales` paths, with no significant alterations reaching the data model. This observation suggests that the developers focused on refining the API structure without requiring extensive modifications to the corresponding data representation.

B. xOpera REST API Evolution

Unlike the SunRocks API, the versioning strategy employed in the xOpera API (Figure 6 left) demonstrates a tendency towards minor version upgrades and patch upgrades. Interestingly, these version upgrades were consistently free of
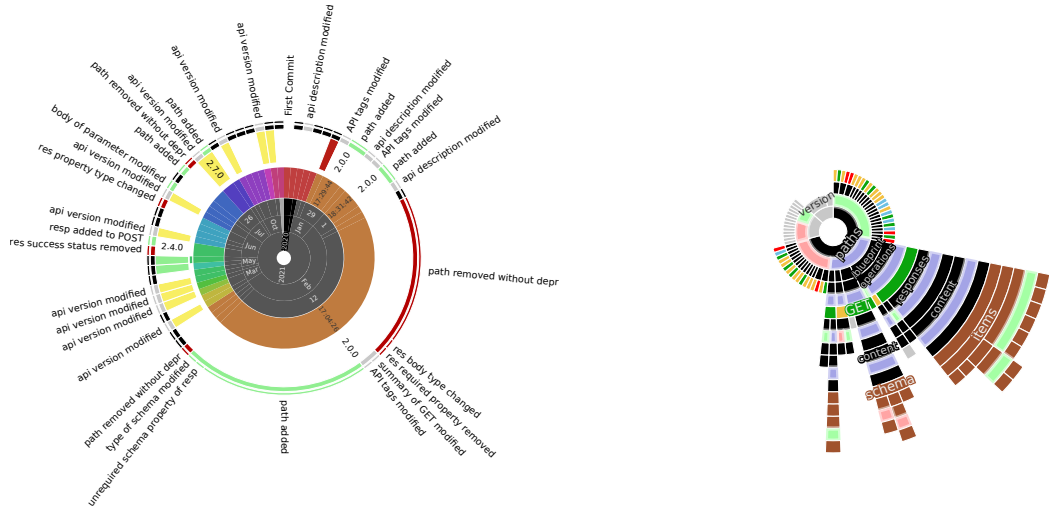
6

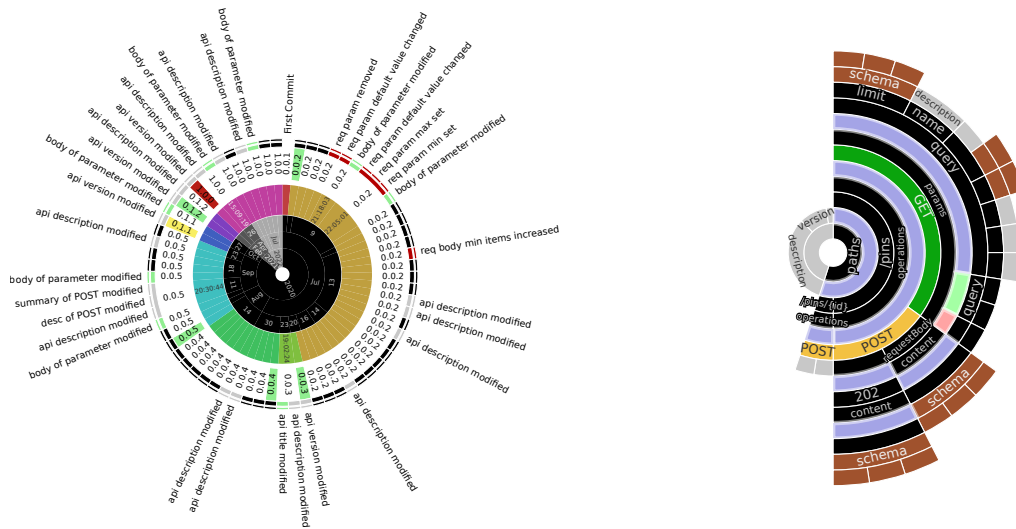Fig. 6. Visualizations of the xOpera REST API Evolution (42 commits over 14 versions during 408 days, 3★)

Fig. 7. Visualizations of the IPFS Pinning Service API Evolution (61 commits over 8 versions during 773 days, 84★)

breaking changes. The largest evolutionary step happened on February 12th, 2021, when 15 paths were removed without deprecation and 17 paths were added. This change was applied to the version 2.0.0 without any immediate impact on the version identifier, which was changed 1 month later to 2.1.0.

The API CHANGES shows that all three types of structural changes occurred with the addition of 25 paths, the modification of 21 paths, and the removal of 17 paths. Additionally, 26 changes impacted the API description metadata. Unlike the SunRocks API, some changes impacted the API data model, e.g., the removal or addition of response schema properties.

C. IPFS Pinning Service API Evolution

The IPFS Pinning Service API (Figure 7) illustrates the early preview release [31] phase of an API, which after almost one year of development reaches version 1.0.0.

Most of the API structure and data model appears to be in place since the beginning, as there are no additions/removals neither at the level of paths or methods, nor at the level of the schema elements, as can be seen from the API CHANGES visualization. 20 out of 22 modifications affected the GET and POST operations of /pins path. The depth query parameter of the GET operation was added and subsequently removed.

Most of the breaking changes are concentrated in the early commits during the first few days of the project, while the remaining commits (largely affecting the natural language documentation) are backward compatible or unclassified. While version identifiers were gradually and regularly upgraded during the pre-release phase, the version 1.0.0 identifier remains fixed, even with minor structural modifications being still applied months after the initial release.

7

Fig. 8. Visualizations of the Xero Projects API Evolution (125 commits over 93 versions during 999 days, 80★)

Fig. 9. Visualizations of the OpenFairDB API Evolution (144 commits over 52 versions during 1563 days, 53★)

D. Xero Projects API Evolution

The Xero Projects API has been selected out of 11 APIs documented in a repository still under active development at the time of writing. Most of the structural changes occurred in the first two months of a 3-year long history. We can see 9 commits, from version `2.8.2` until `2.8.4`, during which 13 paths were removed and 7 paths were added. These breaking changes resulted in a patch version upgrade, up to version `2.8.5`. This version was however reverted back to the `2.8.4` identifier which remained constant until all remaining changes were committed. The version identifier then started to grow all the way to `2.38.0`, resulting in approx 50% of the API CHANGES, due to the co-evolution of this API with the others in the same repository: whenever one API description changes, developers bump the versions of all API descriptions in the same repository.

E. OpenFairDB API Evolution

The OpenFairDB API Evolution (Figure 9) is extracted from a repository in which both the API documentation and the backend implementation are found. Thus, developers will often use patch version upgrades without any API changes to track changes to the underlying backend implementation code. During the summer of 2020, a number of breaking changes were however introduced while keeping the same version `0.8.20` identifier. Moreover, commits of version upgrades never included structural changes.

Overall, there were more paths additions than deletions. And, most of the structural changes at the level of the paths are modifications. The most modified path is the `/search` endpoint (affecting both its response schema and query parameters), while the `/search/duplicates` path was added and later removed from the API.

VI. Discussion

A. API Version Clock

One limitation of this visualization is that it may not scale well when dealing with longer histories of an API. The representation of individual commits and their associated changes can become overwhelming and cluttered, making it challenging to extract meaningful insights from the non-interactive visualization. The interactive version of the visualization can be helpful to zoom into specific sections of the timeline. Alternatively, an icicle plot [48] layout could be used so that the long history of APIs with a large number of commits can be displayed in a scrollable viewport.

To address this limitation and improve scalability, an abstraction technique can be applied to aggregate the changes over longer periods of time in the case of APIs with many frequent commits or long history. For example, all commits leading to a specific version change can be aggregated, considering that developers may first commit changes to the artifact and only update the versioning metadata in a separate commit. This would not fundamentally change the ring structure of the visualization, which would simply result in a less granular commit timestamp ring, where each sector would account for all changes occurring during a certain API version.

The version color on the fourth ring could better reflect semantic versioning. As opposed to uniformly assigning a distinct color to each version identifier, the mapping could use color shade variations for patch versions, similar colors for minor upgrades and different colors for each major release.

Furthermore, a prior investigation into web API versioning practices [43] unveiled that semantic versioning, following the `MAJOR.MINOR.PATCH` format, represents merely the most frequent one out of the 55 diverse API versioning formats. While the current version of the versioning analyzer already detects pre-release tags, it could be further extended to support a broader range of commonly adopted formats, such as calendar-based versioning. In this particular case, the same visualization would show the consistency between the commit timestamp and the version timestamp.

As APIs occasionally change title during their evolution, we plan to enhance the visualization to show during which timeframe each API title was in use. Likewise, it may be useful to filter the commits used to build the visualization based on the specific value given to the API title.

B. API Changes

One of the key design decisions of the API Changes visualization is the lack of precise time information for each individual change. While this enhances scalability by aggregating changes across multiple API versions, it can hinder the ability to analyze the chronological order of changes and understand their possible causal relationships. Given the atemporal nature of the API Changes visualization, it is not possible to perform such analysis with it. To find out when a certain change occurred and which version was affected, the commit corresponding to the selected change can be highlighted in the twin API Version Clock visualization.

The visualization's color scheme can be further enhanced to distinguish other frequently changing API features such as media types, security schemes, or other protocol-specific elements (e.g., response status codes). It would also be possible to apply a color layer to distinguish which changes did break compatibility with clients and which ones did not.

VII. Related Work

A. Visualizing software changes

Alexandru et al. [10] introduce Evo-clocks, a technique that uses circular charts to show changes in software metrics over time. Evo-clocks can reveal patterns and trends in software evolution, such as growth, decay, stability, and volatility.

Another approach to visualize software evolution is presented by Baum et al. [12], who developed GETAVIZ, a framework that generates structural, behavioral, and evolutionary views of software systems. GETAVIZ uses 3D city metaphors to represent the structure and behavior of software entities, and color-coded timelines to show their evolution. Similarly, Pfahler et al. [35] introduces a technique to visualize evolving software using the 3D city metaphor, representing the structure and evolution of software systems. They use height, width, and depth to encode the size, complexity, coupling of software entities, and color to encode their age. They also use animation and morphing to show the changes in software entities over time. Their technique can help developers identify hotspots, outliers, and anomalies in software evolution.

In [16], Hanjalić proposed ClonEvol, a tool that visualizes the evolution of code clones in software systems. ClonEvol uses a mirrored radial genealogy tree to show how clones are created, modified, and removed over time. ClonEvol also provides various metrics and filters to help users analyze the impact of clones on software quality and maintenance.

Kula et al. [23] visualize the evolution of systems and their library dependencies. They use a graph-based representation to show the dependencies between systems and libraries, and a treemap-based representation to show the size and complexity of systems. Also using a treemap-based visualization, Tua et al. [47] presents a technique to visualize software evolution. But in their case, the treemap representations use Voronoi tessellation to partition the space according to the size and complexity of software entities. They use color to encode the age of software entities, and animation and morphing to show their changes over time. However, the visualization does not provide a classification of the visualized changes, which hinders the detection of disruptive changes and the assessment of whether a correct versioning strategy was applied.

B. Sunburst Visualizations

While in this paper we propose to represent key aspects of API evolution using the sunburst visualization, this hierarchical visualization has been successfully applied to many other domains. Liu and Wang [30] proposed a hierarchical information visualization method and applied it to public opinion analysis. They used sunburst visualization to show the distribution of public opinions on different topics and subtopics,

and to compare the opinions across different regions and time periods. Stab et al. [45] presented SemaSun, a visualization tool that uses an improved sunburst visualization metaphor to display semantic knowledge, the structure and content of ontologies, and to support interactive exploration and editing of semantic data. In [42], Rodrigues et al. developed Multi-VisioTrace, a traceability visualization tool that uses sunburst visualization to show the relationships between artifacts in software development. They used sunburst visualization to show the traceability links between requirements, design, code, and test cases, and to support traceability analysis and management. For the same traceability use case, Rodden aimed in [41] to summarize user navigation sequences on websites using a sunburst to show the frequency and order of visits to different pages as well as the transitions between them.

C. Consistency of Versioning and Changes

Previous research work investigated semantic versioning and its impact on software ecosystems. In [37], Raemaekers et al. proposed a metric to measure the stability of software libraries based on the frequency and severity of breaking changes. They applied their metric to 30 Java libraries and found that most libraries had a low stability score. In [38], Raemaekers et al. extended this work by analyzing the Maven repository and comparing the semantic versioning rules with the actual breaking changes detected by bytecode analysis. They found that 14.5% of the releases violated the semantic versioning rules by introducing breaking changes in minor or patch versions. Raemaekers et al. further investigated in [39] the impact of breaking changes on the clients of the libraries and found that 32.9% of the clients were affected by at least one breaking change. In [34], Ochoa et al. replicated the study of Raemaekers et al. on a more recent snapshot of the Maven repository and found that the percentage of semantic versioning violations increased to 22.8%. They also performed a qualitative analysis of the reasons for breaking changes and found that most of them were unintentional or unavoidable.

In the case of NPM, in [36], Pinckney et al. conducted a large scale analysis of semantic versioning in NPM and found that 25% of the releases violated the semantic versioning rules by introducing breaking changes in minor or patch versions. They also found that semantic versioning violations had a negative impact on the adoption and satisfaction of the packages. To mitigate packages versioning inconsistencies on NPM, Abdalkareem et al. proposed in [7] a machine learning approach to determine the semantic versioning type of NPM packages releases based on code changes and commit messages. They achieved an accuracy of 86.7% on a dataset of 5000 releases.

While several empirical studies have investigated web API evolution [13, 20, 26, 44] none of the existing work has specifically examined Web API changes and versions consistency. Our approach can be seen as an initial push towards promoting a more consistent relationship between versioning and functional alterations in this context.

VIII. Conclusion

This paper presents two novel interactive visualizations tailored for developers, researchers, and stakeholders involved in API development, management, and evolution. API Changes and API Version Clock aim to offer valuable insights into the recurrent API changes, and versioning practices, aiding in understanding the evolution and backward compatibility between consecutive API versions and the adherence of the API to semantic versioning practices. The provided visualizations can be integrated into devops pipelines, helping to continuously gather awareness of the entire history of changes and see the evidence needed to enforce the chosen versioning strategies [19, 40]. From the API client developers side, these visualizations can help users to make informed decisions about which API is stable enough to rely on for their system, identify introduced compatibility issues, and reflect on the impact of API modifications on their clients.

The availability of these visualizations through the APICTURE tool provides a valuable resource for API practitioners and researchers, allowing them to explore and analyze API evolution in a comprehensive and intuitive manner. It is released on NPM [1] as command-line tool that automatically generates the API Changes and API Version Clock visualizations from any git repository containing the history of an OAS specification. The tool can render each visualization separately as an SVG or PNG image, but also generate an HTML page with both interactive visualizations, individually or side by side, together with various evolution metrics visualizations and metadata. It can be installed using the command: `npm install -g apict`. We also provide an extended version of the visualizations gallery: APIcture demo [3].

IX. Future Work

We plan to carry out further experiments in order to assess the usefulness, understandability, accessibility, and effectiveness of APICTURE following existing visualization evaluation techniques [33].

While the visualizations have been originally designed in the context of APIs described using the OAS standard, they can be generalized to other artifacts. The API Version Clock requires a stream of commits with the corresponding version identifiers together with metrics characterizing and classifying the changes w.r.t. the previous commit. The API Changes visualization is applicable to show how any nested object structure evolves, as it only requires a lightweight customization for color mapping different properties. We plan to broaden the scope of applicability of the visualization tool by decoupling it from its domain of inception in the near future.

REFERENCES

[1] APIcture. https://www.npmjs.com/package/apict.

[2] Bmore Responsive API. https://codeforbaltimore.github.io/Bmore-Responsive/.

[3] Gallery. https://souhailas.github.io/VISSOFT2023/.

[4] oasdiff tool. https://github.com/Tufin/oasdiff.

[5] OpenAPI Initiative. https://www.openapis.org/.

[6] Semantic Versioning. https://semver.org/.

[7] Rabe Abdalkareem, Md Atique Reza Chowdhury, and Emad Shihab. A machine learning approach to determine the semantic versioning type of npm packages releases. *arXiv preprint arXiv:2204.05929*, 2022.

[8] Juan Pablo Sandoval Alcocer, Fabian Beck, and Alexandre Bergel. Performance evolution matrix: Visualizing performance variations along software versions. In *2019 Working conference on software visualization (VISSOFT)*, pages 1–11. IEEE, 2019.

[9] Carol V Alexandru. *Efficient software evolution analysis: algorithmic and visual tools for investigating fine-grained software histories*. PhD thesis, University of Zurich, 2019.

[10] Carol V Alexandru, Sebastian Proksch, Pooyan Behnamghader, and Harald C Gall. Evo-clocks: Software evolution at a glance. In *2019 Working Conference on Software Visualization (VISSOFT)*, pages 12–22. IEEE, 2019.

[11] Len Bass, Ingo Weber, and Liming Zhu. *DevOps – A Software Architect's Perspective*. Addison-Wesley, 2015.

[12] David Baum, Jan Schilbach, Pascal Kovacs, Ulrich Eisenecker, and Richard Müller. Getaviz: generating structural, behavioral, and evolutionary views of software systems for empirical evaluation. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 114–118. IEEE, 2017.

[13] Fabio Di Lauro, Souhaila Serbout, and Cesare Pautasso. A large-scale empirical assessment of web api size evolution. *Journal of Web Engineering*, pages 1937–1980, 2022.

[14] Murat Erder, Pierre Pureur, and Eoin Woods. *Continuous Architecture in Practice: Software Architecture in the Age of Agility and DevOps*. Addison-Wesley, 2021.

[15] Patric Genfer, Johann Grabner, Christina Zoffi, Mario Bernhart, and Thomas Grechenig. Visualizing metric trends for software portfolio quality management. In *2021 Working Conference on Software Visualization (VISSOFT)*, pages 88–99. IEEE, 2021.

[16] Avdo Hanjalić. ClonEvol: Visualizing software evolution with code clones. In *Proc. First Working Conference on Software Visualization (VISSOFT)*, pages 1–4. IEEE, 2013.

[17] James Higginbotham. *Principles of Web API Design: Delivering Value with APIs and Microservices*. Addison-Wesley, 2021.

[18] André Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stéphane Ducasse. How do developers react to api evolution? a large-scale empirical study. *Software Quality Journal*, 26:161–191, 2018.

[19] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[20] Holger Knoche and Wilhelm Hasselbring. Continuous api evolution in heterogenous enterprise software systems. In *Proc. 18th International Conference on Software Architecture (ICSA)*, pages 58–68, 2021.

[21] Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. Classification of changes in api evolution. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 243–249. IEEE, 2019.

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

[23] Raula Gaikovina Kula, Coen De Roover, Daniel German, Takashi Ishio, and Katsuro Inoue. Visualizing the evolution of systems and their library dependencies. In *2014 Second IEEE Working Conference on Software Visualization*, pages 127–136. IEEE, 2014.

[24] Raula Gaikovina Kula, Ali Ouni, Daniel M German, and Katsuro Inoue. An empirical study on the impact of refactoring activities on evolving client-used apis. *Information and Software Technology*, 93:186–199, 2018.

[25] Patrick Lam, Jens Dietrich, and David J. Pearce. 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*, page 157–179, 2020.

[26] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. A systematic review of api evolution literature. *ACM Computing Surveys (CSUR)*, 54(8):1–36, 2021.

[27] Arnaud Lauret. *The design of web APIs*. Simon and Schuster, 2019.

[28] Deqing Li, Honghui Mei, Yi Shen, Shuang Su, Wenli Zhang, Junting Wang, Ming Zu, and Wei Chen. Echarts: a declarative framework for rapid construction of web-based visualization. *Visual Informatics*, 2(2):136–146, 2018.

[29] 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.

[30] Chanjun Liu and Peng Wang. A sunburst-based hierarchical information visualization method and its application in public opinion analysis. In *2015 8th International Conference on Biomedical Engineering and Informatics (BMEI)*, pages 832–836. IEEE, 2015.

[31] 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.

[32] Mehdi Medjaoui, Erik Wilde, Ronnie Mitra, and Mike Amundsen. *Continuous API management*. O'Reilly, 2021.

[33] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. A systematic literature review of software visualization evaluation. *Journal of systems and software*, 144:165–180, 2018.

[34] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. Breaking bad? semantic versioning and impact of breaking changes in maven central: An external and differentiated replication study. *Empirical Software Engineering*, 27(3):61, 2022.

[35] Federico Pfahler, Roberto Minelli, Csaba Nagy, and Michele Lanza. Visualizing evolving software cities. In *Proc. Working Conference on Software Visualization (VISSOFT)*, pages 22–26, 2020.

[36] Donald Pinckney, Federico Cassano, Arjun Guha, and Jonathan Bell. A large scale analysis of semantic versioning in npm. In *IEEE International Working Conference on Mining Software Repositories*, 2023.

[37] Steven Raemaekers, Arie Van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 378–387. IEEE, 2012.

[38] Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *Proc. 14th International Working Conference on Source Code Analysis and Manipulation*, pages 215–224. IEEE, 2014.

[39] 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.

[40] James Roche. Adopting devops practices in quality assurance. *Communications of the ACM*, 56(11):38–43, 2013.

[41] Kerry Rodden. Applying a sunburst visualization to summarize user navigation sequences. *IEEE computer graphics and applications*, 34(5):36–40, 2014.

[42] Adriana Rodrigues, Maria Lencastre, and A de A Gilberto Filho. Multi-VisioTrace: traceability visualization tool. In *Proc. 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 61–66. IEEE, 2016.

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

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

[45] Christian Stab, Matthias Breyer, Kawa Nazemi, Dirk Burkhardt, Cristian Hofmann, and Dieter Fellner. Semasun: visualization of semantic knowledge based on an improved sunburst visualization metaphor. In *EdMedia+ Innovate Learning*, pages 911–919. Association for the Advancement of Computing in Education (AACE), 2010.

[46] Mirko Stocker and Olaf Zimmermann. From code refactoring to API refactoring: Agile service design and evolution. In *Service-Oriented Computing (SummerSOC 2021)*, volume 1429 of *Communications in Computer and Information Science*, pages 174–193. Springer, 2021.

[47] Davide Paolo Tua, Roberto Minelli, and Michele Lanza. Voronoi evolving treemaps. In *Proc. Working Conference on Software Visualization (VISSOFT)*, pages 1–5, 2021.

[48] Huub van de Wetering, Nico Klaassen, and Michael Burch. Space-reclaiming icicle plots. In *2020 IEEE Pacific Visualization Symposium (PacificVis)*, pages 121–130. IEEE, 2020.

[49] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. Historical and impact analysis of api breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147. IEEE, 2017.

[50] Laerte Xavier, Andre Hora, and Marco Tulio Valente. Why do we break apis? first answers from developers. In *Proc. IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 392–396. IEEE, 2017.

[51] 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.