Visualizing and Exploring Data Access in Microservices Using Interactive Treemaps

Maxime André*, Marco Raglianti†, Anthony Cleve*, Michele Lanza†
*Namur Digital Institute, University of Namur, Belgium
†REVEAL @ Software Institute — USI, Lugano, Switzerland

***Abstract*—The popularity of microservices has grown significantly over the past decade. This architectural style is praised for its ability to ease software evolution, particularly due to the modular, heterogeneous, and dynamic communication nature of microservices. This new way of designing applications has also impacted how databases are integrated. Practitioners generally opt for polyglot persistence, meaning that each microservice manages its own database(s). Decoupling, heterogeneity, and distribution introduce implicit dependencies and multiply data access endpoints. This results in added complexity and challenges in understanding change propagation, which can only be addressed through manual browsing of the codebase, a time-consuming, error-prone, and cumbersome process. A holistic view of such architectures is essential, especially for enabling developers to understand, maintain, and optimize the complex interactions across microservices, particularly from a data perspective.**

We extend a visualization-based approach to support both a high-level view and fine-grained inspection of microservices. Based on static analysis, we generate an interactive treemap for an entire microservices architecture, providing both an overview and the means for more detailed exploration.

We evaluated our approach by assessing the scalability and effectiveness of our visualization. First, we generated interactive treemaps for 10 non-trivial microservices architectures. Then, in a qualitative user study, we asked 6 professional developers to perform specific exploration and understanding tasks (e.g., understanding architectural structure, assessing concept spreading, evaluating technology breakdown, comparing versions, identifying anti-patterns). Our results show that interactive treemaps provide the holistic view needed to aid in evolution tasks.

***Index Terms*—microservices, data access, treemap, software visualization, exploration**

I. Introduction

The popularity of microservices has grown significantly over the past decade. Today, they are adopted by leading companies such as Netflix, Amazon, and Spotify [1], [2]. This architectural style is often contrasted with monoliths. Microservices are characterized by their modularity, heterogeneity, and collaborative interactions. Thanks to these features, such architectures are praised for their ability to facilitate software evolution [1], [2].

At the data layer, this new paradigm has also influenced how databases (DBs) are integrated. Practitioners can now leverage *polyglot persistence* [3], meaning that each microservice is responsible for managing its own DB(s). Since microservices are encouraged to be decoupled, it makes sense to reduce the high coupling on the DB side as well, envisioning similar benefits for maintenance and evolution.

However, recent studies reveal that microservices practitioners still encounter difficulties when evolving microservices architectures, especially from code and data management perspectives [4]–[8]. Decoupling, heterogeneity, and distribution introduce implicit dependencies, multiply data access endpoints, and increase the variety of technologies [9]. Although theoretical principles argue that each microservice and its DBs should be isolated [1], [2], in reality, there are implicit dependencies in the codebases [10], [11]. At the conceptual level, some data entities can be shared across different microservices' DBs, resulting in added complexity and challenges in understanding the impact of changes and their propagation, which, so far, could only be addressed through manual browsing of the codebase. This is a time-consuming, error-prone, and cumbersome process [12] that can benefit from explorable, comprehensive holistic views.

We present an extension of our previous work [9] on visualizing microservices architectures using interactive treemaps. First, we improved the implementation, enabling it to scale to industry-strength systems. Second, we extended the single case study to a thorough evaluation of the treemap visualization, its scalability, and its effectiveness. We generated interactive treemaps for a total of 10 non-trivial microservices architectures. We also conducted a qualitative user study involving 6 professional developers, exploring the data access perspective in microservices architectures through our interactive treemap. Developers were asked to explore and understand a system, commenting on specific aspects of the data access layer (*e.g.,* code fragments, data concepts). The evaluated tasks mimic real-world scenarios such as understanding architectural structure for onboarding new developers, analyzing implementation choices and assessing the impact of changes (and concept spreading) for software evolution, comparing versions and analyzing technology breakdown for feature traceability, and inspecting anti-patterns for bug-fixing.

Our visualization proves to scale to real-world microservices architectures, enabling rapid insight into structure, technology distribution, concept localization, and code quality. A qualitative user study and a quantitative questionnaire (UEQ-S) with 6 developers show that the treemap is effective. All participants completed tasks on systems ranging from 2 to 22 services and up to 1M lines of code. The results indicate "excellent" satisfaction (mean score: 2.1/3.0), with developers stressing the holistic view, efficient navigation, and usefulness for onboarding and codebase analysis.

 DOI: 10.1109/VISSOFT67405.2025.00012

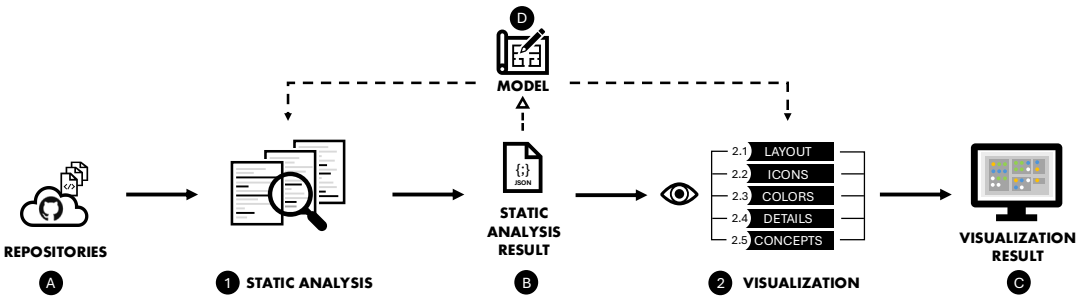

Fig. 1. Approach overview.

II. Interactive Treemap

In software evolution, interactive visualization supports the representation, inspection, and comprehension of codebases. We present an approach, illustrated in Figure 1, to generate interactive treemap visualizations that facilitate the depiction, exploration, and understanding of microservices architectures, with a focus on code fragments performing data access CRUD (Create, Read, Update, or Delete) operations.

Specifically, we mine the microservices architecture of interest using our static analyzer (1) [13], [14]. We download one or more Git repositories (A), parse and query the source code to identify code fragments that represent data access, and extract the relevant information to produce an analysis report (B). The analysis relies on a custom underlying model (D) following a *repository – folder – file – code fragment* hierarchy [13]. We enhance each fragment with additional details, such as the database technology used, the CRUD operation, the specific Object-Relational Mapping (ORM) method employed, and, when available, a sample of the data object or value affected by the operation. We also link them to one or more data concepts, extracted by applying several natural language processing rules (*e.g.,* lemmatization).

We traverse the resulting hierarchical structure recursively to generate the corresponding treemap (2). This layout is well-suited for visualizing hierarchical and recursive large structures in limited 2D spaces [15]–[23]. Each code fragment has a predefined and uniform size and we construct the layout bottom-up. We apply a bin-packing *First Fit Decreasing* heuristic algorithm[1] to pack the rectangles representing the structure. We build upon different aspects of the visualization (*e.g.,* layout, icons, colors, as presented in Figure 1, 2.1–2.5). For instance, the grey-scale nested rectangles represent the containment hierarchy. The darker the color, the more nesting there is, except for the files, rendered brighter to highlight them by contrast. Each code fragment's *operation* in the model determines the icon used to visually represent it. We map the *technology* to a configurable color. Several features bring the interactivity to the treemap. For example, when hovering over a box, a tooltip appears and displays related additional details. A click on a code fragment opens its container source file in the Git repository and shows the code fragment in its original context.

[1] `https://www.npmjs.com/package/bin-pack`

The list of data concepts is also used to interactively highlight related code fragments. The colors are configurable on-the-fly. Finally, the user can zoom in/out and freely navigate. The final result is an interactive treemap (C), as illustrated in Figure 2. Additional details are present in our previous publication [9].

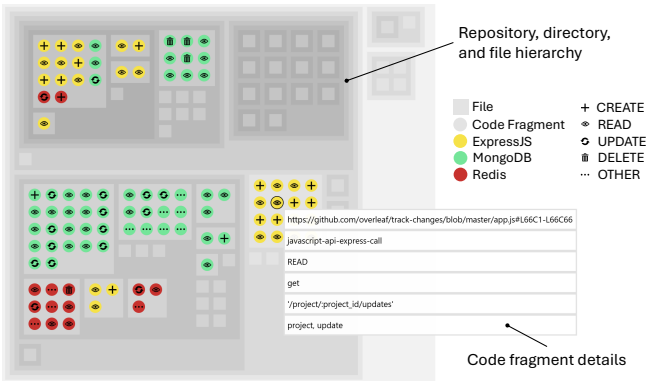

Fig. 2. Visualization result example.

III. Evaluation

Our evaluation is divided into two parts. First, we aim to assess the scalability of our approach across non-trivial, real-world microservices architectures of different sizes and complexities. Then, we conduct a qualitative user study with 6 developers from the industry. All materials and results are available in the replication package described in Section VII.

A. Research Questions

Our goal is to answer the following research questions, by illustrating the scalability of our approach and collecting professional developers' feedback about the effectiveness of our interactive treemap:

- **RQ1 (scalability):** To what extent can an interactive treemap scale to visually represent data accesses in real-world microservices architectures?
- **RQ2 (effectiveness):** How effectively do interactive treemaps meet the needs of microservice developers for data access-related tasks?

TABLE I
SUBJECT MICROSERVICE ARCHITECTURES, SORTED BY DECREASING NUMBER OF SERVICES.

Name	Hash	Description	Services	LoC	Files	Stars	Contributors	Commits
Open Integration Hub	633ca8d	Data synchronization between business apps	22	297k	1,290	190	47	4k
Overleaf	f7e716c	LaTeX online authoring platform	12	698k	5,024	15k	105	24k
Cloudboost	aa45640	Platform for cloud applications	11	545k	1,417	1.4k	23	845
RAIR	1cf4d0f	Infrastructure to create decentralized APIs/apps	5	140k	1,057	1.2k	9	5k
API Table	2632850	Low-code platform to build collaborative apps	4	971k	7,903	14k	66	918
Veniqa	0bee6a8	Customizable e-commerce solution	4	68k	302	1.2k	4	805
Bitpay	ad556e5	Blockchain infrastructure for Bitcoin apps	3	545k	1,227	4.9k	147	16k
PartyDeck	d8f57de	Online card game	3	45k	330	106	1	712
PeerTube	a0c055d	Video streaming platform using Peer2Peer	2	1M	3,444	13.5k	473	14k
Mastodon	2f98134	Microblogging social network	2	532k	8,539	47k	963	18k

B. RQ1: Scalability

We report, based on the generated interactive treemaps, several interesting observations that exemplify the benefits of our approach and how it scales to large and complex systems.

1) Subject Systems: We evaluated our approach with 10 microservices architectures. Table I provides key metrics about the selected systems. To ensure the selection of non-trivial real-world systems, we collected them from GitHub and filtered repositories according to specific criteria. Among these criteria, the targeted systems have been updated at least once since January 1st, 2015, are at least 5 MB in size, have more than 100 stars, and feature a minimum of 100 commits. We limited our dataset to systems written in popular languages for microservices, like JavaScript and TypeScript. We also considered folders and files, especially Docker-Compose and README files, to further understand each microservices architecture and determine whether it was a good candidate. Finally, the authors manually checked the relevance of the selected microservices architectures.

This evaluation aimed to make sure the treemaps generated were scalable while being generated within a few seconds.

C. RQ2: Effectiveness

To assess the usability of our interactive treemap, we conducted a controlled experiment with 6 developers who performed 8 exploration and understanding tasks across 3 microservices architectures (*Open Integration Hub, Cloudboost, Overleaf*). These systems were selected among the 10 subject systems (Section III-B1) for having at least 10 services, a minimum of 250k lines of code (LoC), and at least 20 contributors. We collected developers' qualitative feedback. After briefing them on the context and goals of the experiment, we asked participants to perform 8 tasks leveraging the interactive treemap visualization to gain insights on specific microservices and their architecture (including anti-patterns), database access technologies, CRUD operations, and concepts. Finally, we asked participants to fill out a *User Experience Questionnaire Short (UEQ-S)* [24]–[26] to validate the quality of the implemented treemap visualization. This part aims to assess the effectiveness of our interactive treemap while collecting qualitative developer feedback to identify the synergies among the implemented features.

1) Participants Demographic: The recruited participants belong to 4 different organizations. They have experience in professional software development, ranging from 3 to 10 years, in sectors including R&D, data analytics, consultancy, and banking. Two are students with prior software development experience in the industry, one is a PhD researcher with prior industrial experience, and three are professional developers. They have professional experience with microservices architectures. None of them is an author of this paper. Data about participants and experiment results are completely anonymized.

2) Procedure: The experiment protocol followed the same predefined procedure for each participant, as detailed below. Each individual session lasted from 35 to 60 minutes.

- **Briefing** *(max 10 min)*. The experiment instructor (first author of the paper) begins by explaining the context of DBs in microservices. He also explains in detail the goal of the experiment and the procedure to follow. He gives a brief tutorial on the proposed interactive treemap visualization using a very simple example project: *PartyDeck*. This project was selected among the subject systems. This educational example is light and easy to understand for participants. Then, the experiment instructor provides details on the subject systems. Finally, he reads the understanding and exploration tasks with the participants, ensuring that these are well understood.
- **Exploration and Understanding Tasks** *(max 45 min)*. Participants are asked to explore, one after the other, interactive treemaps of some systems selected among the subject systems. To guide the exploration, developers are asked to perform predefined tasks in the systems while verbalizing their actions. Each sequence is the same for each participant. Task execution time is measured, but there is no time limit. Each task is finally associated with a conclusion status among the following: Success, Fail, or Abandon (after a clear declaration of abandonment by the participant).
- **Note-taking**. The experiment instructor observes and records the actions, interactions, reactions, and overall behavior of the participant. The instructor does not intervene, except in case of technical issues. No help is given. The aim of this step is to qualitatively assess the ease of use and effectiveness of our interactive treemap.

- **Post-session Questionnaire** *(max 5 min)*. After completing the tasks, participants are asked to fill out a short questionnaire. The aim is to quantitatively evaluate the ease of use and effectiveness of our interactive treemap.
- **Closure**. After this step, the participants are free to leave.
- **Interpretation**. The experiment instructor analyzes the results post-experiment.

3) Exploration and Understanding Tasks: The interactive treemap visualization aims to address several needs that we describe with the following use cases:

- **Scalable architectural structure overview**: Provides a scalable, compact 2D treemap layout for a holistic overview of the microservices architecture, highlighting the containment hierarchy of repositories, directories, and files where data access code fragments are located.
- **Leveled exploration**: Proposes an interactive way to explore the system at different levels, with zoom-in and zoom-out capabilities, allowing developers to inspect focused parts with an adjustable granularity of information.
- **Breakdown analysis**: Offers insights into the distribution of technologies, operations, code fragments, and data concepts, highlighting heterogeneity or consistency at macro (entire codebase) and micro (single file) scales. Each breakdown can serve different purposes. For example, technology breakdown can help to justify technological choices and migration strategies, according to consistency or heterogeneity across files and microservices.
- **What-if analyses and change impact assessment**: Supports what-if analyses to assess the impact of changes by highlighting all the components sharing a concept (*e.g.,* data concept modifications, technology switches).
- **Version comparison**: Facilitates comparison between different versions of the system.
- **Component importance evaluation**: Helps to evaluate the importance of each microservice from a data access perspective and identify data-oriented parts of the system and their characteristics (in contrast to parts where the application logic resides).
- **Anti-pattern identification**: Assists in finding and inspecting anti-patterns for refactoring, such as isolated or highly nested code fragments, concept spreading, or excessive technology heterogeneity in a single microservice.
- **Data access details**: Enables fine-grained detail disclosure and inspection, such as localization of specific code fragments in the codebase, the type of data access operation, the method involved and its argument values, and related data concepts (*i.e.,* data entities).
- **In vivo code fragment exploration**: Allows direct jump to the source code files for each code fragment to inspect their context of use or act on them.

We translated these use cases into 8 tasks:

1) Point out and name the microservices of *Open Integration Hub* and *Cloudboost* you identify on the related interactive treemaps.
2) Color and comment on the breakdown of MongoDB and Redis technologies in *Open Integration Hub* and *Cloudboost*.
3) Color and comment on the breakdown of Create, Read, Update, and Delete operations in *Open Integration Hub* and *Cloudboost*.
4) Select the concept "user" and comment on its distribution across files, directories, and services in *Open Integration Hub* and *Cloudboost*.
5) Compare two versions of *Overleaf* and identify a notable change.
6) Point out a strange design choice or anti-pattern that deserves more attention in *Overleaf*.
7) Show the details about a data access code fragment in *Overleaf*.
8) Show the data access code fragment in its real context in *Overleaf*.

These tasks are designed to allow participants to interact with and manipulate the treemap while performing actions relevant to software evolution, especially understanding tasks. They also help developers understand the system structure and data access code fragments. Moreover, they require developers to understand how technologies are implemented. At a finer-grained level, they allow developers to obtain detailed data about data concepts (*i.e.,* data entities) and operations performed. Finally, they provide developers with a way to compare versions, offer insights for anti-pattern identification, and invite them to further explore and analyze the codebases.

4) Questionnaire: After the evaluation session, each participant filled out an anonymous *UEQ-S* questionnaire [24]–[26] to quantitatively assess the usability and effectiveness of our visualization. *UEQ-S* questionnaires are known for their reliability in quickly and efficiently measuring user experience in a quantitative and rigorous way.

IV. Results

A. RQ1: Scalability

1) Scales: In Figure 3, we present an overview of the 10 treemaps generated for the corresponding subject systems (see Section III-B1). These examples demonstrate the scaling of our treemap visualization. The selected microservices architectures describe real-life implementations, ranging from 2 to 22 microservices, from 45k to 1M LoC, from 300 to 8.5k files, developed by 1 to 1k developers across 5k to 24k commits. Some very popular examples have up to 47k stars. While the generation of the static analysis report can take a few minutes, the generation of the treemap takes only a few seconds.

2) Separation of concerns: The layout of our treemap, combined with the visualization of the distribution of technologies or operations, makes it possible to identify data-intensive parts as well as parts devoid of any code fragment. Hovering over some elements can confirm initial observations. The treemap is able to reveal patterns, providing developers with a way to formulate hypotheses and draw conclusions through exploration, meanwhile developing a better understanding of how the architecture is structured.

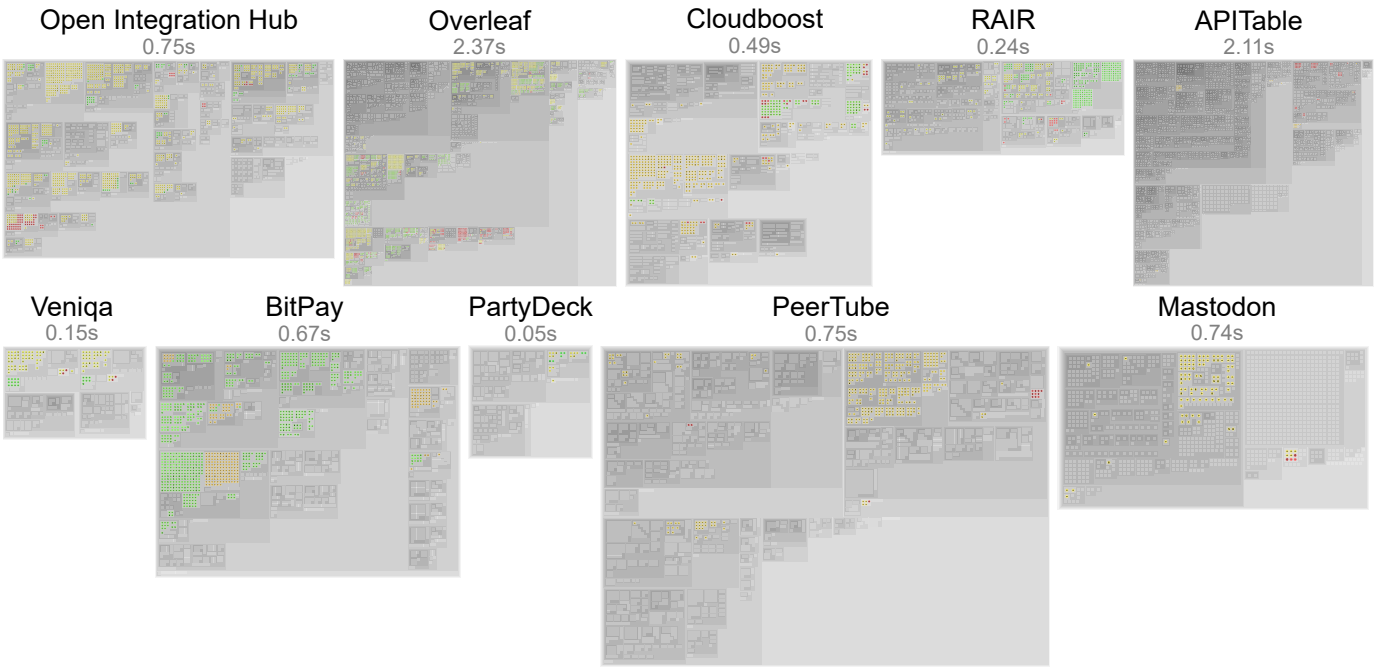

Fig. 3. Treemaps generated from the 10 subject systems, sorted by decreasing number of services.

For instance, one possible task is to determine whether the *separation of concerns* principle is respected. In Figure 4, we can see that *Veniqa* respects this principle by correctly decoupling the responsibilities of backend (top) and frontend (bottom, black frame) microservices.

Fig. 4. Separation of concerns in *Veniqa*, between backend (top) and frontend (bottom) microservices.

3) High nesting: The treemap layout effectively presents the entire repository, folder, and file hierarchy at the same time. According to the state of the practice, treemaps are well suited to the compact visualization of large hierarchical and recursive structures in limited 2D spaces [15], [17]–[23]. This view contributes to the scalability of our visualization, which is capable of adequately representing codebases of hundreds of thousands of LoC, even up to a million for *PeerTube*. In addition, the interactive tooltip reduces potential confusion by instantly indicating the elements hovered over when exploring the hierarchy, from the architecture point of view down to the fine level of a single line of code.

A few examples demonstrate the effectiveness of high-level nesting. In Figure 5, we depict 16 nesting levels identified in *APITable*, another system with almost a million LoC.

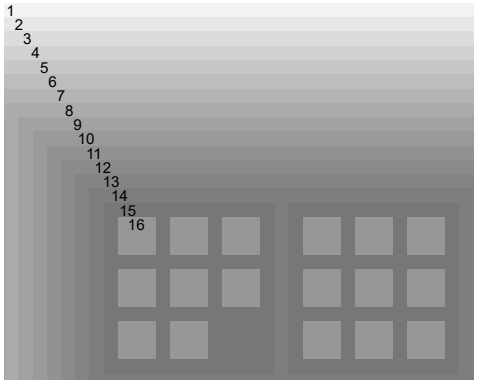

Fig. 5. 16 nesting levels in *APITable*.

4) Technology breakdown: The visualization helps developers understand at a glance the distribution of technologies within the architecture. Figure 6 shows an example on *RAIR*, where we notice two main patterns: Either the technologies are mixed in a heterogeneous way (left), or they are concentrated in a homogeneous way (right). This distribution can be observed at the micro level (*e.g.,* file) or at the macro level (*e.g.,* folder, single microservices, repository). These observations give practitioners a way to rapidly identify the technologies used, assess their prevalence, understand how they are organized, identify polyglot components, localize data-intensive areas, highlight certain files more easily, compare elements within the architecture, etc.

Fig. 6. Distribution of technologies in *RAIR*, heterogeneous on the left and homogeneous on the right.

5) Data Concept Distribution: The interactive treemap visualization enables users to highlight code fragments associated with particular data concepts. This feature helps in assessing how a concept is distributed throughout the system, which is valuable for evaluating the potential impact of modifying that concept. As illustrated in Figure 7 for *BitPay*, the highlighted fragments on the left correspond to the concept *"address"*, which appears in multiple locations. In contrast, the concept *"subscription"*, on the right, is confined to a single, cohesive area of the system, indicating its independence. Consequently, changes in the code manipulating subscriptions or in their data structure (*e.g.,* schema) are easier to implement due to the clear separation of this concept in the architecture.

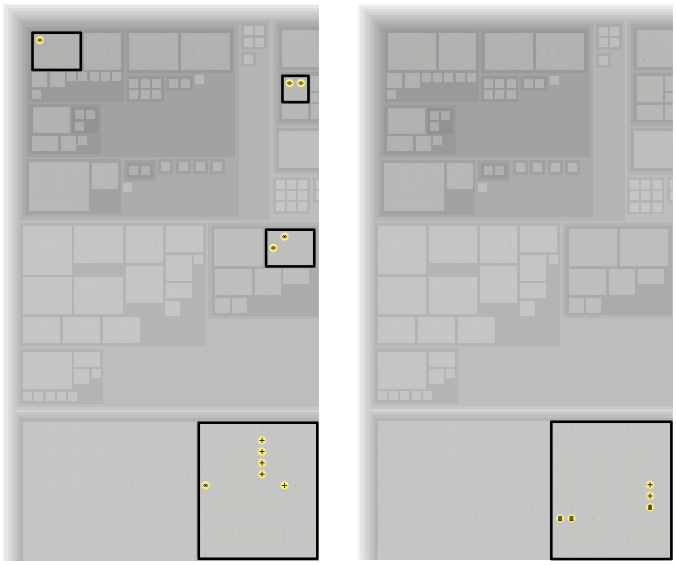

Fig. 7. Concept distribution in *BitPay*, widespread for the "address" concept on the left and concentrated for "subscription" on the right.

6) Version Comparison: The previous analyses considered only a single version of each codebase. However, our approach also facilitates the comparison of different versions of the same project. As shown in Figure 8, we highlight a specific microservice within *Overleaf*, with more than three years separating the two versions displayed.

In the top images, the `/app` component within the `web` microservice remains largely unchanged between versions, as indicated by the similar layout and color patterns, which assist developers in quickly identifying differences. In contrast, the bottom images reveal significant growth and restructuring of the system. Notably, the `/frontend` component was affected by major refactoring, moving data access responsibilities out of the UI, resulting in a cleaner architectural separation.

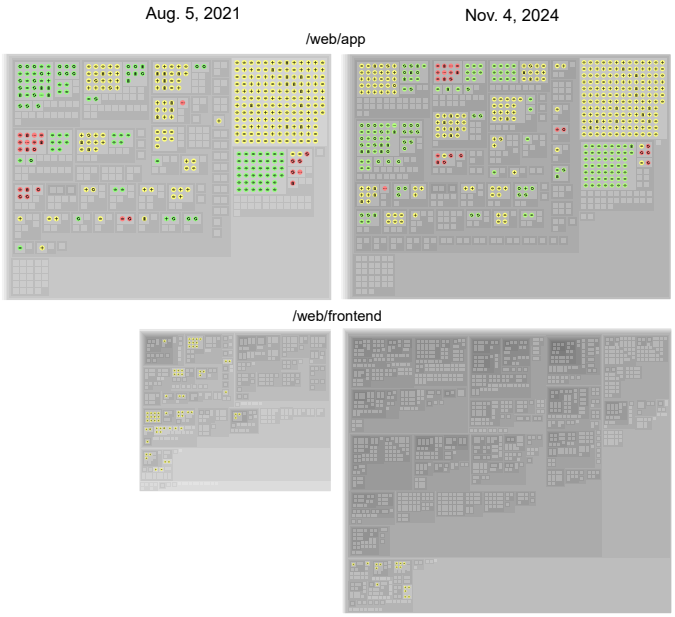

Fig. 8. Comparison of an old version (left) and a new version (right) of an *Overleaf* microservice.

7) Code Fragment Isolation: From a quality assessment point of view, the interactive treemap enables users to quickly spot isolated and deeply nested code fragments within the codebase. While some "distant" fragments may be intentional, they often deserve closer examination as they could indicate potential anti-patterns [27] or residues forgotten after refactoring. Such instances can be challenging to detect manually at these scales, but our visualization eases their identification. Figure 9 illustrates an example of an isolated code fragment in the *Cloudboost* codebase.

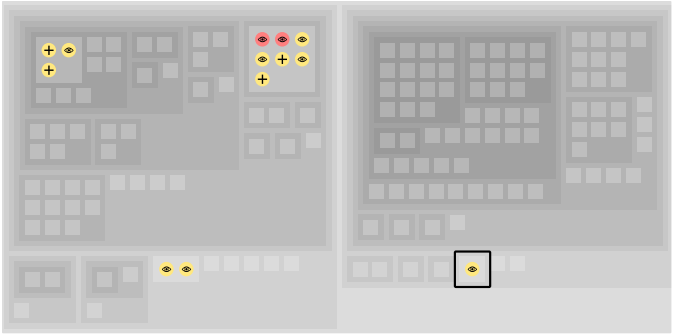

Fig. 9. Isolated code fragment in Cloudboost on the left.

B. RQ2: Effectiveness

1) Developers' feedback: We classified feedback into about twenty conclusions detailed below.

Holistic view. First, participants acknowledged the holistic view and the contained "block" representation as helpful for understanding how the microservices architecture is structured. Some put forward arguments such as *"I'm generally resistant to treemaps, but I have to admit that I like this visualization because we can rapidly identify all the important elements we are looking for in the same place."* and *"The holistic view, especially due to the blocks that group the information. My brain likes it!"*. They also noted, in line with the first research question (Section III-B), the satisfying scaling. One participant said, *"The treemap scales quite well [...], especially in very large codebases where it is very complicated to navigate if we look only at the source files"*.

Navigation. The same participants indicated that the navigation contributes to easier microservices architecture analysis. They gave feedback like *"It's very efficient for code navigation."*, *"I like the navigation including folders, files and code types within the same type. It's easy to get used to."*, *"I like it because it's interactive and we can use only the mouse."*, *"We can navigate inside the entire architecture levels including microservices, folders, and files very quickly"*. Another participant pointed out that *"the treemap helps you find your way at both macro and micro levels"*. On a side note, we noticed that 4 out of 6 developers adopted a top-down navigation approach as suggested by the treemap. The others directly jumped onto the first "attractive" rectangle.

Layout for structure comprehension. In this regard, most practitioners used the hierarchy of shapes and their opacity to understand the structure of the system. Participant 3 said, *"Grey blocks help to get an overview of the structure of the project"*. Participant 6 explained comparing blocks of the same opacity to find similarly nested code fragments (*e.g.,* other potential microservices). Nevertheless, one participant requested a way to filter/hide or change the size of some blocks on demand (*e.g.,* tests) depending on the context of use.

Layout for microservices identification. The layout was appreciated. All participants used the grey rectangles to identify microservices. In the corresponding task, two developers mentioned using fragments as a proxy to identify microservices, since a service is supposed to have at least one data access API endpoint. One participant asked for a way to highlight by coloring or zooming identified microservice blocks.

On-demand details. To ease the identification of specific microservices, all practitioners used the on-demand additional information provided by the tooltip. In particular, the most used feature was the presence of the endpoint URL.

In-context code fragment. Beyond the tooltip, most participants quickly jumped into the codebase by clicking on the code fragment to access its context. They found it intuitive. One participant said, *"The treemap combined with the ability to jump into the code helps to explore further and to understand some exceptions in curious design choices"*. Another noted, *"I like accessing directly the file preview on GitHub"*.

Technology and operation breakdown. Always aiming to go one step further, all developers valued the holistic view of technologies and operations breakdown for spreading, concentration, and prevalence assessment. Two participants noted the possibility to understand the distribution of responsibilities, such as files dedicated to one technology, READ-only (*e.g.,* UI) or DELETE-based microservices (*e.g.,* altering sensitive parts). Another noted that technology breakdown helped to highlight polyglot microservices. Two practitioners mentioned that the variety of operations can be used as a proxy to determine whether a microservice respects the separation of concerns principle. Since each service is supposed to be self-contained, it should contain a certain variety of operations.

Participant 2 explained, *"The treemap is a great help in pinpointing almost instantly where I need to look if I have to work on a problem involving a certain technology"*. Similarly, participant 6 added, *"I know which microservice uses that technology, which is very useful if I have to update this one"* and *"It's useful to see the grouping by operations to assess which service is responsible for reading or writing and therefore modifying the database"*. Participant 4 argued, *"Thanks to the colors, I can quickly identify the top technology"*.

The main ways of highlighting technologies and operations are icons and colors. Most developers appreciated color customizability. Two participants preferred using technology's icon colors, others preferred coloring the type of effect, especially for operations (*e.g.,* red for *delete*, representing danger, blue for *read*, being more neutral). A color-blind practitioner contrasted colors to distinguish code fragments. Another one used colors to distinguish API from DB calls, using the same colors for all DB calls. This participant also matched the colors with Open API standard colors.[2] Finally, two participants used totally random colors. Colors received several suggestions for future work, such as color blindness adaptation, color suggestion, predefined colors, and color preference saving.

Concept highlighting. All practitioners recognized the usefulness of concept highlighting for progressive refinement, helping them understand whether the concept is concentrated or not in an area, to know where to look in case of change, and to get information on related code fragments. Participant 1 said, *"We can see very quickly where the concept is used and its related operations and technologies"*. Participant 3 added, *"Finding concepts helps to find out areas in the code where concepts are used, which is especially important for large codebases"*. They also noted concept lemmatization and said, *"Concept standardization is useful when we want to find a concept based on requirements written in natural language"*. Participant 5 concluded, *"The feature for concept highlighting is the most interesting and it could become indispensable for our company if the feature is taken a step further"*.

Version comparison and structural change identification. In an evolutionary perspective, developers found it easy, thanks to the treemap layout, to identify structural changes, like the restructuring and refactoring of *Overleaf*.

[2] `https://swagger.io/tools/swagger-ui/`

More generally, they all successfully compared both versions, noticing, for instance, the increase of data access calls, nesting levels, components, folders, and files. They were able to localize changed and unchanged parts. Two participants used the visual pattern formed by colored code fragments to compare and match microservices within two versions. Two other participants even noticed the switch from multi-repo to mono-repo that happened in Overleaf a few years ago.[3]

Evolution support. In this context, participant 5 saw the potential of the approach for debugging and evolution tasks, explaining that *"the treemap could be used to support the solving of complex structural problems"*. They added, *"Databases are the pain point in our company. We struggle to get an interlayer overview. This visualization has potential and is aligned with our needs and the market. I like that we can see the data access in the code. It could be very nice to reconcile in the future this view with a deeper view of the data layer,* e.g., *database content. It could help us in debugging"*. Finally, some participants suggested dependency visualization, predefined evolution recommendations, and identification of bad patterns as possible interesting new features.

Quality assessment. Most developers noticed the suitability of the treemap for quickly assessing the system quality and drawing conclusions. They noticed, for instance, DB calls in UI, microservices without any call, microservices that are too large, isolated code fragments, high nesting, anti-patterns to name some of the problems, but also well-structured and well-distributed code fragments. One practitioner explained, *"It also helps to draw conclusions, for instance by pinpointing main design problems. It really makes you think a lot"*. Another added, *"The treemap could be used for code validation, assessing if calls are well-structured and well-placed"*.

Architecture comparison. Sometimes participants went one step further beyond the assigned tasks and found it interesting to compare different architectures. For example, they compared the way *Cloudboost* and *Open Integration Hub* are architected. Some preferred *Cloudboost* and others preferred *Open Integration Hub*. They liked that *Open Integration Hub* puts the services in a dedicated folder while *Cloudboost* mixes them with utils and UI at the root of the repository.

Code fragment isolation. At the reconciliation between macro and micro-level, all developers easily identified isolated code fragments. One said, *"I can see some exceptions. A few code fragments are isolated away from others"*.

Data-intensive localization. Some practitioners focused their analysis on data-intensive zones. Participant 3 argued, *"I like the fact that I can see all the data accesses concentrated just on the databases. It helps me find the areas of code I need to modify when I change my database. The treemap helps me see which microservices can modify the database"*.

Polyglot areas localization. In the same spirit, half of the participants found it interesting to identify polyglot areas of the systems to quickly determine whether some microservices involved more than one database.

[3]`https://github.com/overleaf/overleaf/issues/923`

API and DBs distinction. Alongside databases, four developers put their interest in easily distinguishing API from DB code fragments. One participant said, *"I like that we can easily distinguish API and database calls thanks to the technology colors"* while another specified, *"I can also see where the APIs interact with each other"*.

Separation of concerns. As mentioned, the assessment of separation of concerns particularly interested half of the developers. Nonetheless, all participants were able to rapidly localize side concerns like tests, libraries, utils, UI, etc.

Integration. We noticed a good general affordance on the treemap. Although the scenario of exploration and understanding tasks followed a certain order in the features discovery, and even if the developers discovered the treemap for the first time, almost everyone immediately exploited all the features in an integrated and combined way, showing that they make even more sense when used complementarily.

A participant found it interesting to combine icons with colors to *"mark the eyes"* (*i.e.,* highlight read operations). Another participant used the tooltip in addition to the in-context click to further analyze some code fragments. One also used the colors and icons in combination with the tooltip to analyze and understand the surprising size of a microservice, looking for eventual justifications. One practitioner used colors and concept filter together. He said, *"I like that colors and icons remain when I'm looking at a concept. In that sense, I can see where we create a user in which technology for example"*. Three participants suggested going a step further in the filter combination and extension for finer-grained inspections.

Onboarding. Three developers acknowledged the relevance of the treemap for code onboarding. Participant 2 explained, *"The treemap always gives me the motivation to jump into the codebase. It's a good entry point for code exploration, especially when onboarding a project, and above all on microservices where I sometimes have to consider multiple codebases"*, then adding, *"I'd rather look at the treemap than clone the repo and inspect directories and files, especially when I'm onboarding on a codebase"*.

Understanding. All participants came to the same conclusions for each understanding task. Their opinions were consistent. No critical contradiction was observed, meaning that the treemap is not confusing. All tasks were successfully completed without major struggles or any form of abandonment. Each participant brought interesting ideas and personal comments, often based on their professional experiences. The senior participant concluded, *"The visualization is nice and useful. I'd like to have it at work"*.

Curiosity. Participants did not limit themselves to performing only the requested tasks. They were all curious to explore further. Each of them repeatedly made remarks such as *"That's an interesting way to do it"* or *"This part of the architecture deserves deeper investigation"*, sometimes even diving into the code by clicking on files or code fragments to understand better. The time taken for each task varied greatly because further investigations beyond the evaluation tasks depended on the developer's interest.

The task with the highest average time is *"Version comparison"*. The shortest one is *"On-demand details"*. Overall, tasks lasted from 19 to 44 minutes. Further details on task completion are provided in the replication package (see Section VII).

Interpretation. Generally, practitioners made very similar observations. Nonetheless, they sometimes brought slightly different interpretations. It means that, although guiding the exploration, the treemap visualization does not bias them, tending to conserve a certain neutrality. For example with respect to the preference of structural architectures of two different systems.

Downsides and limitations. Throughout the evaluation, the implementation of the treemap revealed a few downsides. These mainly concerned reasonable UX improvements (*e.g.,* absence of color blind adaptation, color suggestions, zoom sensitivity). On the other hand, two participants noticed from time to time some limitations of the static analysis (*e.g.,* false positives, incomplete information extraction due to the static nature), which was used as a basis to construct the treemap. All suggestions were written down and will be taken into account for future development.

Outlooks. Involved participants suggested some relevant ideas. Some showed interest in visualizing dependencies. Others requested evolution recommendations and bug fixing support. One participant mentioned the need to get more statistics about fragments, technologies, microservices, etc.

Concluding remarks. Overall, all participants shared the same feedback about the features, in terms of satisfaction and discomfort. Generally, developers were mainly satisfied and not confused by features, leveraging one or more features of the visualization to successfully complete the assigned tasks.

2) Questionnaire's results: We conclude with the results of the UEQ-S [24]–[26]. In this questionnaire, participants were asked to rate their overall impression of the treemap, with a particular focus on efficiency and user experience. The evaluation considered two perspectives: pragmatic and hedonic quality, each assessed through 8 sub-criteria. Each sub-criterion was presented as a pair of opposing adjectives and practitioners indicated their perception by selecting a position on a 7-point scale. For instance, they had to choose between "confusing" and "clear" or between "usual" and "leading edge". In addition to the qualitative user study, these data help provide a quantitative perspective, thereby corroborating the developers' claims.

The UEQ-S comes with an analysis tool for interpreting the results. Figure 10 presents the aggregated results obtained from the questionnaire for the six developers. The conclusion indicates an "excellent" overall satisfaction in comparison to the benchmark [25]. Values over 0.8 are considered as a positive evaluation, with a maximum of 3.0 representing the highest score. The overall scale for our treemap is 2.125. Each sub-criterion is associated with mean and variance metrics. It appears that the highest mean is on the "interesting" criterion, which is aligned with participants' curiosity noticed in the feedback. The lowest variance concerns the "clear" criterion, which reinforces our confidence in the effectiveness of the 2D-treemap layout.

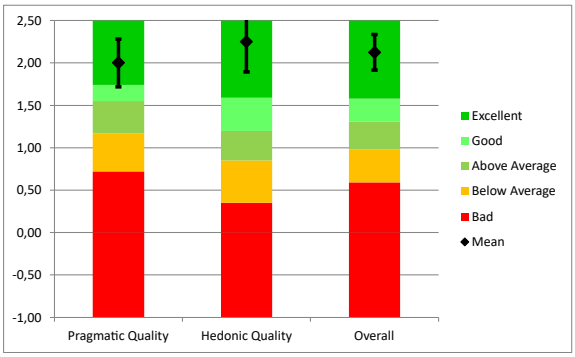

Fig. 10. UEQ-S questionnaire results.

V. Related Work

Several studies have explored holistic visualizations of microservices architectures. They typically focus on interactions between microservices from a software architecture point of view, neglecting a deeper analysis into the code and especially on the data access layer. Moreover, they predominantly rely on directed graph layouts, which can limit interpretability for large systems. Finally, some rely on dynamic analysis, which requires the deployment of the system to perform the proposed analyses and generate their insights. The common thread in the related work is that each approach tries to tame the complexity of the domain, where the intrinsic complexity of software is augmented with the additional complexity of data storage mechanisms. Starting from a simplification approach, our idea follows Ben Shneiderman's mantra: "Overview first, zoom and filter, then details-on-demand" [28]. We use a simple yet sufficient 2D representation (treemaps), making it interactive, and augmenting it with carefully selected meta-information about data access. We comment related works on microservices visualization to position our contribution.

Several authors [4], [6]–[8], [29] perform systematic reviews of techniques that provide holistic views for large systems, involving several independent teams, like microservices architectures. They acknowledge that visualization of them is still a need and emphasize that the absence of overviews could lead to a messy architecture and potential problems in terms of change propagation. These works share the view that providing a way to observe large microservices architectures holistically would help practitioners. Such reviews present several visualization types that we try to broadly categorize. Beyond the prevalent graph-based visualization, there are the chart-based, matrix-based, notation-based, 3D-based, virtual reality and augmented reality-based, metaphor-based, and dashboard-based visualizations. The main tools relying on such visualizations are Amazon's *X-Ray console*,[4] *Simianviz*,[5] *Kiali*,[6] *KubeView*,[7] *MicroART* [30], *Kubernetes Topology Graph*,[8]

[4]`https://aws.amazon.com/xray/`
[5]`https://simianviz.surge.sh/`
[6]`https://kiali.io/`
[7]`https://kubeview.benco.io/`
[8]`https://github.com/kubernetes-ui/topology-graph`

MicroDepGraph [31], *Jaeger UI*,[9] *Zipkin*,[10] *VR-EA* [32], and *ExplorViz* [33].

In the realm of graph-based visualizations, *Ma et al.* introduce an approach for creating graphs that show microservice dependencies for analysis and testing. It helps to detect issues early and track service connections during updates [34]. *Baltes et al.* present a graph-based visualization tool designed to help to understand large industrial service-oriented architectures (based on SOAP or REST APIs). Their aim is to provide companies with a means of monitoring the service landscape, in order to facilitate maintenance decision-making. These challenges are shared with microservices [35].

From a 3D perspective, *Abdelfattah et al.* investigate *Virtual Reality (VR)* for visualizing microservices architectures, especially for representing inter-services dependencies. Based on a controlled experiment, they observe that the third dimension helps developers to identify the dependencies, the cardinalities, and the potential bottlenecks [36]. *Cerny et al.* introduce Microvision [37], a tool able to visualize holistically a microservices architecture. To overcome the issues caused by complex relationships, they opt for *Augmented Reality (AR)*. They point out that the 3D allows leveraging a larger area, thus making it suitable for large codebases, and that the navigation in AR helps to inspect, in a more natural way, various angles and different abstraction perspectives. They evaluate their tool with 6 graduate students performing tasks on real-world applications and giving their feedback through Likert scales.

While joining 3D and the metaphor-based visualizations, *Ardigò et al.* present *M3tricity*, a tool that aims to visualize the evolution of software systems, with particular emphasis on the data and information access layer. Their tool supports program comprehension and evolution analysis tasks [38], [39].

Some opt for dashboard-based approach gathering different visualizations. *Mayer et al.* propose a dashboard for visualizing dependency graphs and charts of static and dynamic data from microservices architectures. They evaluate their approach with 15 practitioners combining surveys and interviews [40]. Bringing runtime data to the table, *Silva et al.* present *μViz*, a graph-based visualization tool using runtime traces to provide four views: logical (service connections), physical (Docker or K8s instances), trace (runtime traces), and workflow (aggregated traces). These views are integrated in a dashboard with several charts. However, this contribution is a mockup [41]. *Manglaras et al.* also investigate the challenges of understanding the structure of microservice architectures systems due to their distributed nature. They surveyed practitioners to identify key problems, finding that they need auto-generated documentation as interactive visualization (especially helpful if in parallel with the source code). They also suggest the use of trace visualizations [42]. In a follow-up work, they propose *MicroKarta*, a dashboard addressing some needs emerged from their previous studies, for which they strengthened the requirements thanks to industrial feedback [43].

[9]`https://github.com/jaegertracing/jaeger-ui`
[10]`https://zipkin.io/`

Finally, beyond the works on microservices, other software visualizations extend the treemap layout adding new properties, like, for example, *Cushion treemaps* [16] and *ViewFusion* [44]. While considering runtime data, some authors also introduce heat maps and trace visualizations [45]–[47].

VI. Conclusion

Microservices architectures have become popular for building scalable and evolvable applications. However, modularity and heterogeneity introduce significant challenges for understanding codebases, especially when facing polyglot data persistence. Existing approaches neglect the data access layer, opt for trivial visualizations, or require dynamic analysis.

We presented an interactive treemap visualization aiming to support both a high-level view and fine-grained inspection of microservices and of their data accesses. The main contribution of this paper is the approach evaluation, conducted on realistic, large, and modern systems and through a qualitative user study involving 6 professional developers. Our results indicate that the treemap visualization (1) scales to real-world systems, and (2) is effective for onboarding, comprehension, and analysis of microservices codebases. Developers highlighted the numerous advantages of its holistic view, intuitive navigation, and usefulness for inspecting technology breakdowns and concepts or performing quality assessment.

Looking forward, we plan to enhance our approach with advanced evolution analysis and recommendation techniques. A comparison with alternative state-of-the-art visualizations specifically tailored for microservices would be also relevant to assess accuracy and speed advantages of our simple yet powerful and scalable interactive visualization.

VII. Replication Package

To ensure verifiability and replicability of our work, the artifacts of our evaluation are publicly available as open source at: https://figshare.com/s/a19f8aa13fea21ecf79c

Acknowledgments

This research is supported by the SofinaBoël Fund for Education and Talent and the Federation Wallonie-Bruxelles (FWB), as part of the ARC project RAINDROP, and by the Swiss National Science Foundation (SNSF) through the project "FORCE" (SNF Project No. 232141).

References

[1] C. Richardson, *Microservices Patterns: with Examples in Java*. Simon and Schuster, 2018.
[2] S. Newman, *Building microservices*. O'Reilly Media, Inc., 2021.
[3] J. Lewis and M. Fowler. (2014) Microservices. [Online]. Available: https://martinfowler.com/articles/microservices.html
[4] T. Cerny, A. S. Abdelfattah, V. Bushong, A. Al Maruf, and D. Taibi, "Microservice Architecture Reconstruction and Visualization Techniques: A Review," in *IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2022, pp. 39–48.
[5] T. Cerny and D. Taibi, "Static Analysis Tools in the Era of Cloud-native Systems," in *International Conference on Microservices (Microservices)*, 2022, arXiv preprint. [Online]. Available: https://arxiv.org/abs/2205.08527
[6] M. E. Gortney, P. E. Harris, T. Cerny, A. Al Maruf, M. Bures, D. Taibi, and P. Tisnovsky, "Visualizing Microservice Architecture in the Dynamic Perspective: A Systematic Mapping Study," *IEEE Access*, vol. 10, pp. 119 999–120 012, 2022.

[7] G. Parker, S. Kim, A. Al Maruf, T. Cerny, K. Frajtak, P. Tisnovsky, and D. Taibi, "Visualizing Anti-Patterns in Microservices at Runtime: A Systematic Mapping Study," *IEEE Access*, vol. 11, pp. 4434–4442, 2023.

[8] T. Cerny, A. S. Abdelfattah, J. Yero, and D. Taibi, "From Static Code Analysis to Visual Models of Microservice Architecture," *Cluster Computing*, pp. 1–26, 2024.

[9] M. André, M. Raglianti, A. Cleve, and M. Lanza, "Understanding Data Access in Microservices Applications Using Interactive Treemaps," in *IEEE/ACM International Conference on Program Comprehension (ICPC): ERA Track*. IEEE/ACM, 2025.

[10] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, "Data Management in Microservices: State of the Practice, Challenges, and Research Directions," *VLDB Endowment*, vol. 14, no. 13, pp. 3348–3361, 2021.

[11] M. André, "Automated Database Schema Evolution in Microservices," in *Conference on Very Large Data Bases (VLDB): PhD Workshop Track*, vol. 3452. CEUR-WS, 2023, pp. 37–40.

[12] A. Lercher, J. Glock, C. Macho, and M. Pinzger, "Microservice API Evolution in Practice: A Study on Strategies and Challenges," *Journal of Systems and Software*, vol. 215, p. 112110, 2024.

[13] M. André, E. Rivière, and A. Cleve, "Data Access-centered Understanding of Microservices Architectures," in *IEEE International Conference on Software Architecture (ICSA): NEMI Track*. IEEE, 2025.

[14] M. André, E. Rivière, and A. Cleve. DENIM Reverse Engineering. Zenodo. [Online]. Available: https://doi.org/10.5281/zenodo.14740539

[15] B. Shneiderman, "Tree Visualization with Tree-maps: 2-d Space-filling Approach," *ACM Transactions on Graphics*, vol. 11, no. 1, pp. 92–99, 1992.

[16] J. J. Van Wijk and H. Van de Wetering, "Cushion Treemaps: Visualization of Hierarchical Information," in *Symposium on Information Visualization (InfoVis)*. IEEE, 1999, pp. 73–78.

[17] M. Balzer and O. Deussen, "Exploring Relations within Software Systems Using Treemap Enhanced Hierarchical Graphs," in *IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 2005, pp. 1–6.

[18] M. Balzer, O. Deussen, and C. Lewerentz, "Voronoi Treemaps for the Visualization of Software Metrics," in *ACM Symposium on Software Visualization (SoftVis)*. ACM, 2005, pp. 165–172.

[19] R. V. Hees and J. Hage, "Stable and Predictable Voronoi Treemaps for Software Quality Monitoring," *Information and Software Technology*, vol. 87, pp. 242–258, 2017.

[20] E. F. Vernier, A. C. Telea, and J. Comba, "Quantitative Comparison of Dynamic Treemaps for Software Evolution Visualization," in *Working Conference on Software Visualization (VISSOFT)*. IEEE, 2018, pp. 96–106.

[21] W. Scheibel, C. Weyand, and J. Döllner, "EvoCells—A Treemap Layout Algorithm for Evolving Tree Data," in *International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP)*, vol. 3: IVAPP, 2018, pp. 273–280.

[22] W. Scheibel, M. Trapp, D. Limberger, and J. Döllner, "A Taxonomy of Treemap Visualization Techniques," in *International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP)*, vol. 3: IVAPP, 2020, pp. 273–280.

[23] D. P. Tua, R. Minelli, and M. Lanza, "Voronoi Evolving Treemaps," in *Working Conference on Software Visualization (VISSOFT)*. IEEE, 2021, pp. 1–5.

[24] B. Laugwitz, T. Held, and M. Schrepp, "Construction and Evaluation of a User Experience Questionnaire," in *Symposium of the Austrian HCI and Usability Engineering Group (USAB)*. Springer, 2008, pp. 63–76.

[25] M. Schrepp, A. Hinderks, and J. Thomaschewski, "Construction of a Benchmark for the User Experience Questionnaire (UEQ)," *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 4, no. 4, pp. 40–44, 2017.

[26] M. Schrepp, A. Hinderks *et al.*, "Design and Evaluation of a Short Version of the User Experience Questionnaire (UEQ-S)," *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 4, no. 6, pp. 103–108, 2017.

[27] B. A. Muse, M. Rahman, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, "On the Prevalence, Impact, and Evolution of SQL Code Smells in Data-Intensive Systems," in *International Conference on Mining Software Repositories (MSR)*. ACM, 2020, pp. 327–338.

[28] B. Shneiderman, "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations," in *The Craft of Information Visualization*. Elsevier, 2003, pp. 364–371.

[29] D. G. Balreira, T. da Silva Araújo, and F. Petrillo, "Visualizing kubernetes distributed systems: An exploratory study," in *Working Conference on Software Visualization (VISSOFT)*. IEEE, 2023, pp. 12–22.

[30] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-based Systems," in *IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 298–302.

[31] M. I. Rahman, S. Panichella, and D. Taibi, "A Curated Dataset of Microservices-based Systems," in *Joint of the Summer School on Software Maintenance and Evolution (SSSME)*. CEUR-WS, 2019, pp. 1–9.

[32] R. Oberhauser and C. Pogolski, "VR-EA: Virtual Reality Visualization of Enterprise Architecture Models with ArchiMate and BPMN," in *International Symposium on Business Modeling and Software Design (BMSD)*. Springer, 2019, pp. 170–187.

[33] F. Fittkau, A. Krause, and W. Hasselbring, "Software Landscape and Application Visualization for System Comprehension with ExplorViz," *Information and Software Technology*, vol. 87, pp. 259–277, 2017.

[34] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, "Using Service Dependency Graph to Analyze and Test Microservices," in *IEEE Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2018, pp. 81–86.

[35] S. Baltes, B. Pfitzmann, T. Kowark, C. Treude, and F. Beck, "Visually Analyzing Company-wide Software Service Dependencies: An Industrial Case Study," in *Working Conference on Software Visualization (VISSOFT)*. IEEE, 2023, pp. 23–27.

[36] A. S. Abdelfattah, T. Cerny, D. Taibi, and S. Vegas, "Comparing 2D and Augmented Reality Visualizations for Microservice System Understandability: A Controlled Experiment," in *IEEE/ACM International Conference on Program Comprehension (ICPC)*. IEEE, 2023, pp. 135–145.

[37] T. Cerny, A. S. Abdelfattah, V. Bushong, A. Al Maruf, and D. Taibi, "Microvision: Static Analysis-based Approach to Visualizing Microservices in Augmented Reality," in *IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2022, pp. 49–58.

[38] S. Ardigò, C. Nagy, R. Minelli, and M. Lanza, "Visualizing Data in Software Cities," in *IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 2021, pp. 145–149.

[39] ——, "M3triCity: Visualizing Evolving Software & Data Cities," in *ACM/IEEE International Conference on Software Engineering (ICSE): Companion Proceedings*. ACM, 2022, pp. 130–133.

[40] B. Mayer and R. Weinreich, "An Approach to Extract the Architecture of Microservice-based Software Systems," in *IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE, 2018, pp. 21–30.

[41] S. Silva, J. Correia, A. Bento, F. Araujo, and R. Barbosa, "μ Viz: Visualization of Microservices," in *International Conference Information Visualisation (IV)*. IEEE, 2021, pp. 120–128.

[42] O. Manglaras, A. Farkas, P. Fule, C. Treude, and M. Wagner, "Problems in microservice development: Supporting visualisation," in *Working Conference on Software Visualization (VISSOFT)*. IEEE, 2023, pp. 62–72.

[43] ——, "MicroKarta: Visualising Microservice Architectures," in *International Conference on the Foundations of Software Engineering (FSE)*, 2024, pp. 607–611.

[44] J. Trümper, A. C. Telea, and J. Döllner, "ViewFusion: Correlating Structure and Activity Views for Execution Traces," in *Theory and Practice of Computer Graphics*. The Eurographics Association, 2012, pp. 45–52.

[45] O. Benomar, H. Sahraoui, and P. Poulin, "Visualizing Software Dynamicities with Heat Maps," in *Working Conference on Software Visualization (VISSOFT)*. IEEE, 2013, pp. 1–10.

[46] A. Krause, M. Hansen, and W. Hasselbring, "Live Visualization of Dynamic Software Cities with Heat Map Overlays," in *Working Conference on Software Visualization (VISSOFT)*. IEEE, 2021, pp. 125–129.

[47] D. Vandamme, H. Sahraoui, and P. Poulin, "Understanding High-Level Behavior with a Light-Traces Visualization Metaphor," in *Working Conference on Software Visualization (VISSOFT)*. IEEE, 2021, pp. 140–144.