Visualizing Data Access Traces in Microservices Using Animated Heat Treemaps

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

***Abstract*—Microservices have become a prevalent architectural style over the past decade, emphasizing the modular and dynamic nature of heterogeneous and distributed units that communicate with each other. Moreover, they promote polyglot persistence, meaning that each microservice is responsible for managing its own database(s), often with heterogeneous technologies. One of the downsides is the increase of the number and diversity of data access endpoints and exchanges. Additionally, the decomposition introduces implicit dependencies that affect code and data understanding and co-evolution. Maintaining a comprehensive high-level view of this kind of architecture is challenging, yet essential for software evolution tasks. Previous works have already proposed holistic representations and visualizations of data access in microservices. However, these are mainly based on structural and fixed snapshots, neglecting the dynamic perspective.**

We present an approach to enhance static visualizations. First, we record data-access-centered execution traces in microservices architectures through a static analysis-based refinement of dynamic instrumentation. Then, we replay scenarios over an existing static treemap, animating the sequence of data accesses and highlighting hotspots in the codebase through time. Our contribution, the animated heat treemap, helps developers to understand how data management operates inside microservices. We validated our approach on Overleaf, a popular online collaborative LaTeX authoring platform, with a real-world scenario. We discuss the results obtained and provide insights and reflections.

***Index Terms*—microservices, dynamic traces, data access, visualization, animated heat treemap**

I. Introduction

Microservices architectures have gained popularity since their appearance at the beginning of the past decade, now being widely adopted by large companies such as Amazon, Google, Netflix and Spotify [1], [2]. They are known for quality attributes such as *modularity*, *heterogeneity*, and *interoperability* [3]. Although these characteristics are supposed to ease software evolution, recent studies show that they hinder obtaining an overview of the system [4]–[6].

Moreover, the *polyglot persistence* [7], the multiple heterogeneous databases (DBs) distributed across different microservices, introduce an additional layer of complexity. For instance, the same concept could be shared across multiple microservices and their DB(s). Despite the logical decoupling of microservices, some explicit dependencies can become implicit [8], with conceptual changes made in one DB leading to change propagation issues [9]. Developers must ensure that changes are *well-propagated* (*i.e.,* completely and consistently) in all the related parts of the system. This is difficult, especially when systems are large and complex, and when the developer is not familiar with the codebase.

Previous works have proposed visualization techniques to provide developers with a holistic view of microservices architectures [2], [4], [6], [8]. These techniques typically rely on either static or dynamic analysis, but rarely on both. In a data management context, visualizations based exclusively on static analysis offer an overview of distributed data accesses, but fail to provide information regarding the actual usage of these accesses.

We present an approach to visualize data access through time for an execution scenario. We run a heuristic-based static analysis [3] to find data accesses. Then, we play a scenario over an instance of the system, instrumented to log dynamic information at runtime, driven by the static analysis. Finally, we combine static and dynamic information to animate the sequences of data accesses, highlighting the hotspots. The visualization is presented as an interactive video player, allowing the developer to play, pause, stop, explore, and navigate through the animation on the treemap [8]. In addition, the developer can display the previous and incoming calls, visualize the data access sequence, enable a "realistic time scale" or visualize the heat map within a given time window.

Visualizing data access hotspots through time over a given scenario allows developers to identify the most important code fragments, helping them understand how data management operates inside microservices. This can help in identifying potential bottlenecks in distributed microservices architectures, and is crucial for prioritizing co-evolution of data representation and handling code.

II. Animated Heat Treemaps

Our contribution builds upon the static treemap visualization by *André et al.* [8], integrating dynamic data access information, as depicted in Figure 1. In that paper, they propose an interactive treemap aiming to present, in a 2D compact space, a holistic view of microservices architecture. They use nested rectangle shapes to represent the repository – folder – file – code fragment hierarchy. Each code fragment represents a data access in the source code. Features like zoom in/out, navigation, hover tooltip, concept highlighting, and colorization participate in the understanding.

Over a software repository (A), we run a static analysis (1) [3] to retrieve data access-related code fragments (B). The static analysis result (B) and its underlying model (D) are provided as an input to the dynamic analysis (1'). Then, we play a scenario over the instrumented instance of the system that was previously statically analyzed.

©2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1109/VISSOFT67405.2025.00017

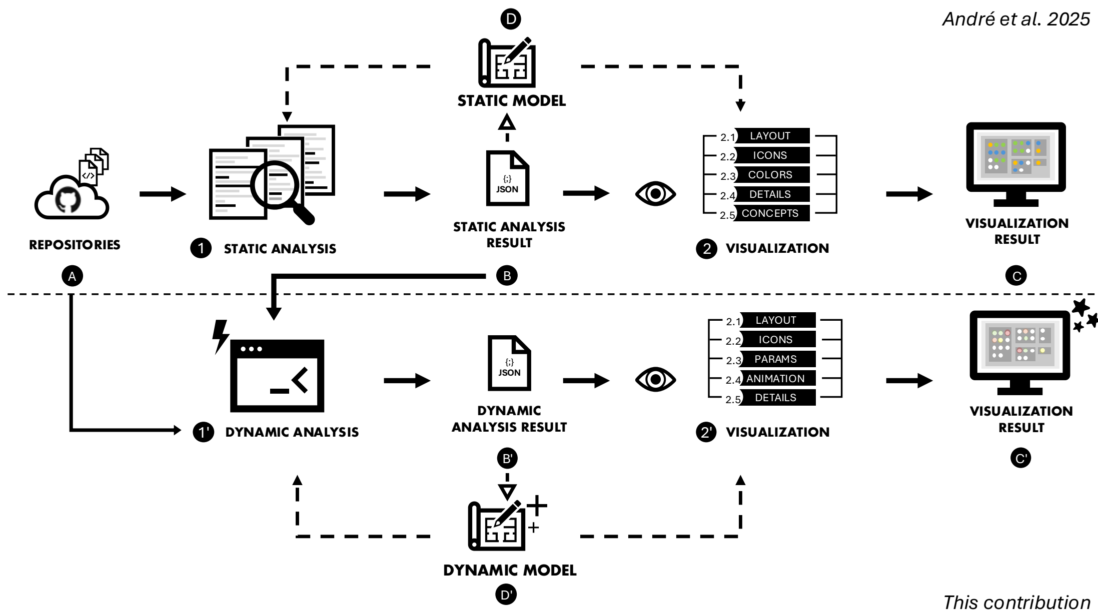

Fig. 1. Approach overview. Partly derived from [8] (top).

The traces collected during the dynamic analysis step (B') populate the dynamic model (D'), which is derived from the static model (D). We instantiate our dynamic interactive visualization (2') with its properties, based on the static interactive treemap (C) and the scenario's trace in (D'), to visualize the resulting animated heat treemap (C').

A. Capturing Traces

We present a case study (Section III) analyzing services written in JavaScript. We use *NodeProf*, a dynamic analysis framework for Node.js [10], to collect execution traces. At each function invocation, we check whether the location was identified by the static analysis as a data access location. We record the timestamp of the function call and the source code location data. Then, we use the log file containing the traces as input to expand our static analysis model, adding the data for each recorded call within the corresponding code fragment.

B. Heat Visualization and Trace Animation

The visualization incorporates animation controls inspired by a video player (Figure 2), comprising control buttons to *play*, *pause*, and *adjust playback speed*, and a *scrub* bar, allowing the user to jump to specific calls in the animation.

Within the animation, the invoked code fragments are denoted by colored circle filters that range from green to red according to the frequency of calls during the execution scenario. Hues towards green reflect less frequent use, while those towards red indicate more frequent use. The size of the circle filters also increases proportionally with the frequency of calls to the fragment, reinforcing the perception of hotspots.

Since the visualization is interactive, the user can hover over code fragments, displaying static and dynamic information. When the animation is paused, users can inspect recent (previous) and upcoming calls, leading to or stemming from the current one (Figure 3).

The heat treemap offers three animation options:

- *Call sequence*, highlighting the sequential flow of calls within a specified timeframe.
- *Realistic time scale*, playing the animation with realistic delays based on the recorded intervals between calls, *fast forwarding*[1] delays exceeding the user-defined threshold.
- *Calls time window*, highlighting only those calls that occur within a user-defined time window, starting from the most recent one.

III. Case Study

Our case study concerns Overleaf, a popular, open source, and collaborative LaTeX authoring platform. Its GitHub repository[2] has 15k stars and 106 contributors. We used the version with commit hash `f94adbf`, from March 2025. It contains 700k lines of code among 865 directories and 5k files. The data access layer counts 2.3k code fragments across 11 out of 12 microservices, mixing different data access technologies (*e.g.,* MongoDB, Redis).

We collected execution traces on a self hosted and instrumented instance of Overleaf (as described in Section II), used for writing a first draft of this very same paper. The experimental scenario consisted in one author performing actions (*e.g.,* editing, uploading figures, commenting) on the project over 6 days, resulting in a total of 109,165 calls.

There are two relevant aspects of dynamic data access traces emerging from the visualization: Hotspots and call sequences.

[1] Fast forward substitutes pauses above a set threshold with a pre-defined amount of time to "compress" the idling periods.

[2] `https://github.com/overleaf/overleaf`

Fig. 2. Overview of the user interface.

Previous Calls

1344	services/real-time/app/js/DocumentUpdaterManager.js {} Lines 134 → 152	09/05 14:22:18,858
1345	services/real-time/app/js/DocumentUpdaterManager.js {} Lines 143 → 150	09/05 14:22:18,866
1346	services/document-updater/app/js/DispatchManager.js {} Lines 44 → 85	09/05 14:22:18,882

Current Position

1347	services/document-updater/app/js/RealTimeRedisManager.js {} Lines 40 → 40	09/05 14:22:18,892

Upcoming Calls

1348	services/document-updater/app/js/RealTimeRedisManager.js {} Lines 41 → 41	09/05 14:22:18,893
1349	services/document-updater/app/js/RealTimeRedisManager.js {} Lines 42 → 46	09/05 14:22:18,894
1350	services/document-updater/app/js/RealTimeRedisManager.js {} Lines 47 → 51	09/05 14:22:18,894

Fig. 3. Previous and incoming calls information.

A. Hotspots

The service showing the greatest concentration of hotspots is *document-updater* (Figure 4), which is responsible for *"applying incoming updates to documents in real-time"*.[3]

Writing a paper involves performing small changes to a document, each triggering one or more data access calls. The most frequently called code fragments are *reading*, *updating*, and *other*[4] operations on Redis DBs. The red fragments (in `RedisManager.js` and `RealTimeRedisManager.js`) were called more than 5,000 times.

[3]https://github.com/overleaf/overleaf/tree/f94adbf/services/document-updater/README.md

[4]Any non-CRUD operation like, for example, *"multi"* or *"exec"* for Redis.

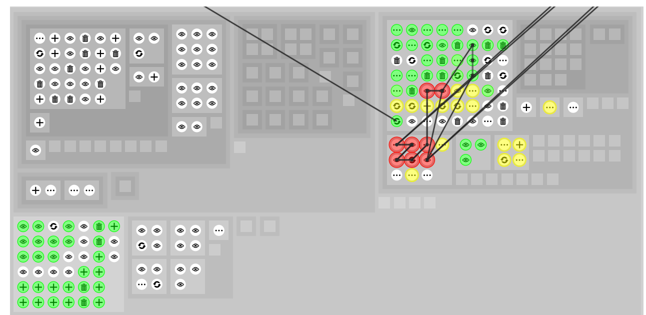

Fig. 4. *document-updater* service hotspots.

The *document-updater* service plays a crucial role in Overleaf's data management, with its Redis DB used to support collaborative editing. Visualizing the frequency of these data accesses provides valuable insights about which components should be prioritized during maintenance and optimization efforts. Data accesses in the hotspots raise architectural questions: Using a single database instance can represent a bottleneck under increased load, especially within a multi-user context. This information could also guide decisions about appropriate scaling strategies.

B. Calls sequences

In Figure 5, we show three moments in the evolution of the animated heat treemap for our paper writing scenario. We also provide a video of the visualization along with the traces and an image of the final state of the animation in our replication package (see Section VII).

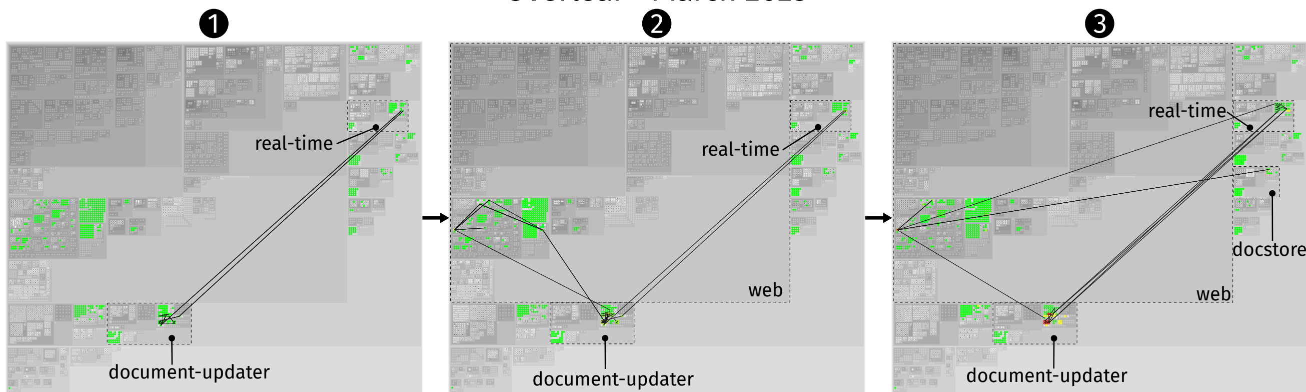

Fig. 5. Evolution of the animated heat treemap over time, with the sequence tracing enabled.

The lines connecting the code fragments represent the *call sequence* within a specified time window (5 seconds in this example). This option reveals repeated temporal data access patterns, indicating that specific code fragments are frequently invoked together (in sequence). We can thus identify data access trends in the scenario.

In the first snapshot (Figure 5, (1)), we see how an external call triggers a sequence of calls between data access fragments, mostly inside *document-updater*. The second and third snapshots ((2) and (3)) show the evolution of the call chain in the 5 seconds time window, involving other microservices (*document-updater*, *web*, *real-time* or *docstore*) during the document update operation performed in the analyzed execution scenario. This sequential pattern reveals how closely related operations are triggered successively during typical user interactions. The configurable trace time window offers flexibility in analyzing shorter and longer temporal traces, depending on the targeted system and scenario.

The proximity of calls in the same microservice, potentially to the same DB, emphasizes potential bottlenecks but also cohesion, while large connections show orchestration and interdependencies between microservices. Further analysis with the interactive treemap could lead to insights on the referenced concepts, for example, for change impact analysis [8].

IV. Discussion

The Overleaf case study summarizes some of the possibilities enabled by our dynamic instrumentation approach and the heat treemap visualization. The traces appear as connected sequences and the spatial positioning in the treemap provides contextual information about the parts of the systems (the different microservices) interacting. The temporal chain can be explored step by step or the overall picture of the heat treemap can be leveraged for higher level observations.

While effective for capturing execution traces, the instrumentation comes with limitations.

Our experiments revealed significant performance overhead when instrumenting the entire system, particularly in the *web* service. This slowdown is partly due to our approach checking every function invocation at runtime, which creates a noticeable computational burden that reduces service responsiveness.

Alternative dynamic analysis techniques can be considered, like, for example, automatic static insertion of logging instructions in the code before, after, or around data access function and method invocations [11], as identified by the static analysis rather than at runtime. Some cases would require attention while using this approach, for example, when data access invocation occurs within return statements or inside conditional expressions. Such instrumentation could be implemented with relative ease in some languages, especially those which support aspect-oriented programming, such as AspectJ or Spring AOP in Java environments. These frameworks provide clean separation of cross-cutting concerns, like separating logging from business logic, offering a more efficient alternative to our dynamic instrumentation approach while maintaining similar observability benefits.

Finally, we performed a case study scenario with a single user to mitigate a limitation of our trace reconstruction method. With multiple users there is no guarantee for two subsequent calls to be directly connected (*i.e.,* not triggered independently by the concurrent users). While implementation concurrency can have similar effects, we argue that, in our scenario, the traces are still relevant since they are triggered by a single action. Otherwise, the instrumentation would need to reconstruct the source of the call (*e.g.,* from the stack trace).

Future Work: Our current work focuses on JavaScript microservices. We plan to support other programming languages to better fit the heterogeneous nature of microservices and evaluate the generalizability of our approach.

We also plan to add new features. We will develop metrics and visualizations for database queries performance, error rates, and access frequency to schema elements.

For what concerns the visualization, we intend to explore alternative 2D layouts based on interaction patterns between code files, where the distance between components is determined by the frequency of joint or successive accesses, potentially revealing functionally related entities through proximity.

V. Related Work

Similarly to our work, *Krause et al.* presented an approach using the software city metaphor with heat map overlays to visualize metrics based on live traces analysis [12], inspired by *Benomar et al.* [13]. *Vandamme et al.* proposed an interactive animated visualization based on a light-traces metaphor to explore execution scenarios on Java programs [14].

Cerny et al. reviewed the state of the art of microservices architectural reconstruction techniques based on static and dynamic analysis [4]. For visualization based on static analysis, there are tools such as *MicroDepGraph* [15], *MicroArt* [16], and *IslandViz* [17]. Dynamic analysis tools, such as *OpenTelemetry*, *Jaeger*, *Amazon X-Ray Console*, and *Simianviz* [4], leverage graph-based views, with directed acyclic graphs or topological representations. Similarly, *Gortney et al.* conducted a systematic mapping study of system architecture visualization with dynamic analysis [2]. Among the dynamic analysis techniques, they mention *process mining*, *distributed tracing* and *monitoring*, complemented by visualization tools (*e.g., MicroArt* [16], *Dapper* [18], *Jaeger*[5], *Zipkin*[6], *μViz* [19]).

Parker et al. conducted a mapping study on visualizations of anti-patterns in microservices at runtime [20]. They compared visualization tools leveraging runtime traces to build dependency graphs (*e.g., MAIG* [21]) and reconstruct microservices architecture (*μTOSCA* and *μFreshener* [22]).

VI. Conclusion

We presented an approach that enhances static analysis-based visualization with dynamic information. The resulting visualization, the animated heat tree map, highlights the evolution of the calls frequency of the code fragments through time. This new visualization offers users a comprehensive view of data access patterns across multiple services during program execution. We evaluated our animated heat tree map using a case study on Overleaf with a realistic scenario. The results shows that the visualization is effective in helping to identify the data access hotspots and patterns during the execution of the program. Moreover, it allows users to quickly pinpoint which code fragments are responsible for high data access frequencies, enabling them to optimize their code accordingly.

VII. Replication Package

The artifacts of our case study are publicly available as open source at https://figshare.com/s/f24ccdd180e3cbf7810f

Acknowledgments

This research is supported by the Erasmus+ programme of the European Union, the SofinaBoël Fund for Education and Talent, and the ARC project RAINDROP from the Federation Wallonie-Bruxelles (FWB). We also thank the Swiss National Science Foundation (SNSF) for the financial support through the project "FORCE" (SNF Project No. 232141).

[5]https://www.jaegertracing.io
[6]https://zipkin.io

References

[1] C. Richardson, *Microservices Patterns: with Examples in Java*. Simon and Schuster, 2018.

[2] 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.

[3] M. André, E. Rivière, and A. Cleve, "Data access-centered understanding of microservices architectures," in *International Conference on Software Architecture (ICSA): Companion Proceedings*. IEEE, 2025.

[4] T. Cerny, A. S. Abdelfattah, V. Bushong, A. Al Maruf, and D. Taibi, "Microservice architecture reconstruction and visualization techniques: A review," in *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] 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 *International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2022, pp. 49–58.

[7] J. Lewis and M. Fowler. (2014) Microservices. [Online]. Available: https://martinfowler.com/articles/microservices.html

[8] M. André, M. Raglianti, A. Cleve, and M. Lanza, "Understanding data access in microservices applications using interactive treemaps," in *International Conference on Program Comprehension (ICPC)*, 2025.

[9] 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*, vol. 14, no. 13, pp. 3348–3361, 2021.

[10] H. Sun, D. Bonetta, C. Humer, and W. Binder, "Efficient dynamic analysis for node.js," in *International Conference on Compiler Construction (CC)*, 2018, pp. 196–206.

[11] A. Cleve, N. Noughi, and J. Hainaut, "Dynamic program analysis for database reverse engineering," in *Generative and Transformational Techniques in Software Engineering (GTTSE)*, vol. 7680. Springer, 2011, pp. 297–321.

[12] 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.

[13] 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.

[14] 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.

[15] M. I. Rahman, S. Panichella, and D. Taibi, "A curated dataset of microservices-based systems," in *Joint Proceedings of the Summer School on Software Maintenance and Evolution*. CEUR-WS, 2019.

[16] 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 *International Conference on Software Architecture Workshops*. IEEE, 2017, pp. 298–302.

[17] A. Schreiber, L. Nafeie, A. Baranowski, P. Seipel, and M. Misiak, "Visualization of software architectures in virtual reality and augmented reality," in *Aerospace Conference*. IEEE, 2019, pp. 1–12.

[18] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010.

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

[20] 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.

[21] I. U. P. Gamage and I. Perera, "Using dependency graph and graph theory concepts to identify anti-patterns in a microservices system: A tool-based approach," in *Moratuwa Engineering Research Conference (MERCon)*. IEEE, 2021, pp. 699–704.

[22] J. Soldani, G. Muntoni, D. Neri, and A. Brogi, "The μTOSCA toolchain: Mining, analyzing, and refactoring microservice-based architectures," *Software: Practice and Experience*, vol. 51, no. 7, pp. 1591–1621, 2021.