A Case for Microservices Orchestration Using Workflow Engines

Anas Nadeem  
anas.nadeem@ndsu.edu  
North Dakota State University  
Fargo, North Dakota, USA

Muhammad Zubair Malik  
zubair.malik@ndsu.edu  
North Dakota State University  
Fargo, North Dakota, USA

ABSTRACT
Microservices have become the de-facto software architecture for cloud-native applications. A contentious architectural decision in microservices is to compose them using choreography or orchestration. In choreography, every service works independently, whereas, in orchestration, there is a controller that coordinates service interactions. This paper makes a case for orchestration. The promise of microservices is that each microservice can be independently developed, deployed, tested, upgraded, and scaled. This makes them suitable for systems running on cloud infrastructures. However, microservice-based systems become complicated due to the complex interactions of various services, concurrent events, failing components, developers’ lack of global view, and configurations of the environment. This makes maintaining and debugging such systems very challenging. We hypothesize that orchestrated services are easier to debug and to test this we ported the largest publicly available microservices’ benchmark TrainTicket [24], which is implemented using choreography, to a fault-oblivious stateful workflow framework Temporal [19]. We report our experience in porting the code from traditional choreographed microservice architecture to one orchestrated by Temporal and present our initial findings of time to debug the 22 bugs present in the benchmark. Our findings suggest that an effort towards making a transition to orchestrated approach is worthwhile, making the ported code easier to debug.

1 INTRODUCTION
Service-oriented applications are increasingly becoming cloud-native and are built as a collection of small, independent, and loosely coupled microservices [8, 11]. Large web companies such as Tencent, Uber, Netflix, and Airbnb are increasingly building their core business systems using microservice architecture [24]. The promise of microservices is that each microservice can be independently developed, deployed, tested, upgraded, and scaled. This makes them suitable for systems running on cloud infrastructures. However, these benefits come at a cost and microservice-based systems become complicated due to complex interactions of various services, events happening concurrently, components failing, developers’ lack of global view, and configurations of the environment. This complexity and dynamism of microservice systems pose unique challenges for system developers and makes them hard to implement and debug.

Using Temporal workflow engine, programmers write their code in a familiar language while the platform orchestrates the tasks and makes the application resilient to all failures. This means any state
within the workflow is durable and frees up the developer to focus on the business logic of the application rather than spend most of the time building resilience. Code written for Temporal needs to adhere to certain constraints, for example, a purely random variable within part of the code that needs to re-execute, can break stateful-workflow assumptions: executions must be idempotent and deterministic. These are semantic errors that will break part of the code which otherwise will normally be considered valid. We implemented a simple linter \(^1\) to deal with such issues and our experience in porting code from original microservices bugs benchmark to temporal framework was fairly smooth.

We make the following contributions: a) We port the largest publicly available microservice benchmark to a fault-oblivious stateful workflow engine and report our experience; b) We experimentally evaluate debugging of 22 bugs present in the TrainTicket benchmark using debugging process supported by Temporal; c) We compare our results with experimental results on original benchmarks and present our observations and insights to provoke further research. The experiments are released as an open-source project \(^2\) and can be replicated with a simple set-up.

2 TRANSITIONING TO TEMPORAL

Moving to the Temporal framework requires a paradigm shift to writing code in comparison to the traditional methods of developing microservice systems. While it hides the complexity of handling various edge cases resulting in a resilient and fault-tolerant system, transitioning to Temporal framework-based system requires some refactoring. The framework requires the separation of business logic from control logic. Subsequently, at a high level, every business process is accomplished through orchestrating a workflow. All the interaction between microservices must be managed by Temporal. Any non-deterministic action that is prone to failure must be wrapped by an activity. These activities and workflows are finally registered with a worker which is responsible for picking these tasks from a queue within Temporal, and finally executing these tasks. To better understand the idea of how to transition from a traditional interaction between several microservices to Temporal, we illustrate mapping such interaction in the benchmark system and employ a similar strategy to map all the business processes within TrainTicket to workflows and activities.

2.1 Background: Benchmark System

To systematically study how Temporal constructs, workflows, activities and workers, map to a realistic microservice system, we port TrainTicket \([24]\) to Temporal. This system provides typical train ticket booking functionalities such as purchase, cancellation, and changing by making use of over 30 fine-grained microservices written in multiple languages including Java, NodeJS, Python, and Go. In addition to that, TrainTicket incorporates 22 fault cases collected from an extensive industrial survey in order to serve as a benchmark system for microservices-based research. We deploy this benchmark system along with the Temporal server on a machine with a 20GB available memory, in the form of Docker containers. After mapping the benchmark system, we study the role of Temporal in 1) assisting

\(^1\)https://github.com/arise-ndsu/temporalint

\(^2\)https://arise-ndsu.github.io/trainticketworkflows

![Figure 1: original cancelTicket process within TrainTicket](image1)

![Figure 2: cancelTicket workflow after porting to Temporal](image2)
the worker construct serves as the baseline task distribution mechanism within Temporal. Therefore, we register all of our functional entities including the cancelTicket workflow, and all the activities with a CancellationWorker. Figure 2 illustrates the cancelTicket process mapped into Temporal constructs.

3 DEBUGGING FAULTS IN TEMPORAL

To illustrate the debugging process within Temporal, we consider the same workflow that we discussed in the previous section and follow a similar strategy for the rest of the faults. We use the benchmark system with a fault injected within the cancelTicket workflow. To inject this fault case, TrainTicket slightly modifies the system to lack strict sequence control and simulate network congestion while invoking the inside-payment-service for the refund functionality and order-service to set the order status to cancelled. With a simulated network congestion, the refund through inside-payment-service gets triggered after the order-service has already set the status to cancelled, due to which inside-payment-service fails to proceed with the request. After mapping the same workflow within, we invoke the drawBackMoneyActivity and setOrderCancelledActivity asynchronously to replicate the scenario within the cancelTicket workflow. We highlight the steps that were involved in debugging this fault within the Temporal ecosystem, in the context of a formal debugging model [15].

- Problem Space Construction: The problem space construction was a development step in the debugging process and involved developing an initial understanding of the fault. We observe that for the fault under consideration, Temporal Web UI [21] produces a stack trace, as well as a visual trace graph, illustrated in Figure 3 and Figure 4, respectively. We collect and observe these logs and traces to develop a preliminary understanding of the fault.
- Identification of Fault Symptoms: This step involved setting up an environment to reproduce the fault. Since Temporal makes business processes explicit in terms of a workflow, we quickly identified the microservices involved in the workflow. Moreover, this straightforwardness of the workflow also helped us speed up the environment setup by enabling us to create a minimalist version of the environment to include only the services participating in the workflow. After the setup, we execute the fault case to evaluate the system outputs and compare them with the expected outputs to identify discrepancies.
- Fault Diagnosis: In this step, we use our knowledge gained in the prior steps to hypothesize the location of the fault. Through thorough analysis using the debugging functionality on Temporal Web UI, we identify the root cause and the precise location of the fault. With the help of the holistic trace view of the workflow on Temporal from its invocation to where it incurred the fault, we reached the exact location within the cancelTicket workflow which triggered the fault. Analyzing the visual trace in Figure 4, we observe that setOrderCancelledActivity completes prior to drawBackMoneyActivity which breaks the intended expectations.
- Solution Generation and Verification: The final steps involved fixing the identified fault. The fault was corrected by adding the necessary sequence control after which the system automatically resumed its execution and resulted in the completion of the workflow. Final steps involved the execution of necessary tests in order to verify that the fault was resolved.

3.1 Preliminary Findings

Table 1 sums up our preliminary findings from a user case study. The table reports the debugging time by our user, of the replicated fault-cases from the benchmark system against the time required to debug the same fault in the original TrainTicket study. User in our study had more than two years of industry experience with microservices development and had no prior knowledge of the faults. However, the user went through elementary training of available debugging support on Temporal. We note that by utilizing the mixture of debugging capabilities offered by Temporal, the time required to debug these fault cases was lesser than the overall time taken in the original study. We also found that Temporal self-contains necessary tools to support debugging and testing of the system, in comparison to the microservice-benchmark paper, which had to employ external tools for the debugging process.

Consistent with the fault types in the original benchmark system, we discuss how Temporal responded to each category of faults. The faults in the benchmark system broadly fall into the categories: Internal faults are caused as a result of implementation in the microservice itself. Interaction faults are caused as a result of interactions between various microservices. Environment faults occur as a result of misconfiguration of the infrastructure [24].

Temporal plays no significant role in the debugging of faults that are internal in nature, as the cause of these faults mainly lies in misinterpreted requirements and typically do not cause any failure within the system. In the case of faults in interaction of microservices, Temporal fully preserved the state of the execution. The execution was resumed by Temporal automatically as soon as the bug was resolved. Similarly, upon the occurrence of an environment fault due to a service downtime or service unavailability, Temporal managed to fully preserve the current execution state and resumed execution as soon as the unavailable microservice came back up.

4 RELATED WORK

Microservices is one of the fastest-growing areas in software engineering [4], however, there is limited high-quality research being done in the field. Hassan et al. [10] considered design trade-offs for microservices along the dimensions of size/number and the global/local
non-functional requirement satisfaction. More recent work [9] has focused on qualitative analysis of composing microservices by choreography or orchestration. However, to the best of our knowledge, we are the first to present a quantitative comparison of compositional choices on a single benchmark. Zhou et al. [25] provide a good literature review of publicly available benchmarks of microservices and informed our choice for TrainTicket. Debugging microservices has gained much attention recently. Our work is highly inspired by a detailed survey, benchmark and empirical study by Zhou et al. [24]. They recorded experiences and processes of practitioners of varied skill and experience levels, systematically recreated the most common bugs experienced by practitioners and performed a detailed study of commonly used debugging techniques and effort required for each one of them. This became the basis of our work and comparison. Hearthiadi et al. [12] provided a framework for systematic testing of failure handling capabilities of microservices. Service calls, graphs, service usage logs, and attribute graphs have been used recently [16, 23] for root cause analysis of anomaly detection and debugging of availability issues. Temporal provides highly detailed logs and call histories that can benefit from such analysis.

5 DISCUSSION

Choreography: Promise vs Reality: Loose service coupling and strong cohesion in a choreographed microservices architecture seem very promising for agility and fault tolerance. It makes adding and removing services as simple as connecting or disconnecting a service from an appropriate channel in the event broker. Loose coupling also implies that choreography isolates microservices, such that if one service fails, other services not dependent on it can carry on while the issue is rectified. Choreographed, event-driven microservices allow for development teams to operate more independently and focus on their key services. Once these services have been created, they are now easily able to be shared between teams. However, teams that have built larger systems learn that much technical debt is incurred in the process [17]. Process flows are “embedded” within the code of multiple services. This replication of code is a headache for maintenance. Also, there are tight coupling and assumptions around input/output and other service level agreements that make it harder to adapt to changing needs. Further, many critical system-level questions cannot be answered immediately, such as: “How much are we done with process X?”

A Lesson from History of Computing: In the early days of information systems, practically everybody wrote their own data store and spent a significant portion of their project time managing data management code, which was cumbersome and error-prone [18]. It was followed by standardized libraries and hierarchical systems until Codd presented the relational model of data [3], and Chamberlin and Boyce introduced A Structured English Query Language [2]. We envision that workflow engines will do for microservices what Relational Database Systems did for information systems. We argue that workflow engines can relieve developers from focusing on low-level distributed programming concerns (such as implementing ACID constraints) and enable them to focus on implementing business logic.

Fault-oblivious Stateful Programming: Temporal Server handles the durability, availability, and scalability of the application. In terms of CAP theorem [7], each server instance is eventually available and highly consistent. In effect, Temporal Server provides a durable virtual memory per workflow execution, that is not linked to any specific process. It preserves the full application state (including program stacks with local variables) across all kinds of software and hardware-related failures. Temporal SDK builds on these capabilities and enables users to write their application code using the full power of the programming language. Temporal ensures that a triggered call will never fail, thus leading to the utility of long-running code spanning over multiple days or even months (let’s say a method that needs to perform something after every 7 days).

6 CONCLUSION

In this work, we have ported the TrainTicket benchmark reflecting typical faults of a microservice to a novel fault-oblivious stateful workflow orchestration engine Temporal. We have used the replicated faults to assess the time to debug the faults in orchestration-based microservice implementation using Temporal’s Web UI that supports stack tracing and visual tracing. We observed that orchestration makes debugging easier and faster. In the future, we plan to run the presented study with more participants, port other benchmarks to Temporal, and study its impact on availability issues.

7 ACKNOWLEDGEMENTS

We are extremely grateful to Mr. Samar Abbas for his insightful discussions and technical support for Temporal.
REFERENCES

[1] Apache. 2021. Airflow. Available: https://airflow.apache.org/.
[2] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A Structured English Query Language. In Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, USA, May 1-3, 1974, 2 Volumes. 249–264. https://doi.org/10.1145/800296.811515
[3] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. Commun. ACM 13, 6 (1970), 377–387. https://doi.org/10.1145/362384.362685
[4] Mordor Intelligence Corp. 2021. Cloud Microservices Market - Growth, Trends, COVID-19 Impact, and Forecasts (2021 - 2026). Available: https://www.mordorintelligence.com/industry-reports/cloud-microservices-market.
[5] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In OSDI’04: Sixth Symposium on Operating System Design and Implementation. San Francisco, CA, 137–150.
[6] Apache. 2021. Airflow. Available: https://airflow.apache.org/.
[7] A. Fox and E.A. Brewer. 1999. Harvest, yield, and scalable tolerant systems. In Proceedings of the Seventh Workshop on Hot Topics in Operating Systems. 174–178. https://doi.org/10.1007/HOTOS.1999.798396
[8] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. 2017. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In 2017 IEEE International Conference on Software Architecture, ICSEA 2017, Gothenburg, Sweden, April 3-7, 2017. IEEE Computer Society, 21–30.
[9] Mahtab Haj Ali. 2021. Measuring the Modeling Complexity of Microservice Choreography and Orchestration: The Case of E-commerce Applications. Ph. D. Dissertation. Ottawa, ON, CA. Advisor(s) Benyoucef, Morad.
[10] Sara Hassan and Rami Bahsoon. 2016. Microservices and Their Design Trade-Offs: A Self-Adaptive Roadmap. In 2016 IEEE International Conference on Services Computing (SCC). 813–818. https://doi.org/10.1109/SCC.2016.113
[11] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. In 36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016. IEEE Computer Society, 57–66.
[12] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. MicroRCA: Root Cause Localization of Performance Issues in Microservices. In NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium. 1–9. https://doi.org/10.1109/NOMS47738.2020.9110353
[13] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. IEEE Trans. Software Eng. 47, 2 (2021), 243–260.
[14] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Poster: Benchmarking Microservice Systems for Software Engineering Research. In 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion). 323–324.