2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C) | 978-1-6654-9493-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSA-C54293.2022.00029

A Saga Pattern Microservice Reference Architecture for an Elastic SLO Violation Analysis

1st Sandro Speth
Institute of Software Engineering
University of Stuttgart
Stuttgart, Germany
sandro.speth@iste.uni-stuttgart.de

2nd Sarah Stieß
Institute of Software Engineering
University of Stuttgart
Stuttgart, Germany
sarah.stiess@iste.uni-stuttgart.de

3rd Steffen Becker
Institute of Software Engineering
University of Stuttgart
Stuttgart, Germany
steffen.becker@iste.uni-stuttgart.de

***Abstract*—Reference architectures are becoming increasingly popular for industry and researchers as benchmark solutions to test their novel concepts and tools. While many reference architectures exist in the microservice domain, they are often not built on state-of-the-art technologies. Furthermore, many existing reference architectures do not use lightweight and asynchronous communications, such as messaging, do not have out-of-the-box self-adaptation and do not consider state-of-the-art microservice patterns. Therefore, this paper proposes a self-adaptive microservice reference architecture that implements the microservice saga pattern. The architecture is implemented in Java Spring Boot and uses the Eventuate Tram framework for the saga orchestration. Moreover, the architecture is instrumented to export performance metrics for monitoring and data for system-wide tracing to check for correct execution of the system and its adaptations. The objective of this reference architecture is to provide a benchmark for explaining self-adaptation and propagation of service-level objective (SLOs) violations across an architecture with complex patterns. In addition to the architecture, we provide defined SLOs and load profiles to stress the architecture.**

***Index Terms*—Microservices, Reference Architecture, Saga Pattern, Service-Level Objectives, Self-Adaptation**

I. Introduction

Reference architectures are sample architectures to showcase various concepts in the particular domain they reflect [1]. By using a reference architecture, developers get architectural support and guidance for the systematic development of systems in their respective domain [2]. Furthermore, software engineering researchers and practitioners increasingly use reference architectures as benchmark solutions to conduct repeatable studies, test novel concepts and tools and evaluate the effectiveness and limitations of their work [3].

Among the plethora of reference architectures that exist in the microservice domain [4], many reference architectures disregard state-of-the-art technologies and patterns. Moreover, as most reference architectures were implemented for a specific purpose, because no other architecture fulfilled the requirements, most of them were not maintained after fulfilling that purpose. Therefore, only a few reference architectures kept up to date with the rapidly developing technologies. Furthermore, while most reference architectures implement some patterns, they usually restrict themselves to a limited set of frequently used ones, such as the API-Gateway [5] [6]. Thus, they seldom implement lesser-known patterns [7], for example, the saga pattern [5], [6] which handles long-living tasks and their rollback in case of a failure. As a result, there is a lack of reference architectures to benchmark novelties on a wider range of patterns. Although microservices should communicate in lightweight asynchronous ways [8], many existing reference architectures rely on synchronous calls, e.g. only providing REST over HTTP APIs. Furthermore, having out-of-the-box self-adaptation is crucial for realistic modern cloud-native scenarios. Especially if a tool or technology needs to be tested for self-adaptive systems, it is critical to have a benchmark system that implements out-of-the-box autoscaling. Even though reference architectures need to allow fully automated deployment of new instances to enable autoscaling [6], most existing works do not provide fully automatable provisioning. Thus, they cannot provide out-of-the-box self-adaptation.

Therefore, we propose the *T2-Project*, a self-adaptive microservice reference architecture that (1) implements multiple patterns, including the less common and more complex saga pattern [5], (2) provides out-of-the-box self-adaptation, (3) uses state-of-the-art technologies and frameworks to implement the patterns, such as the Eventuate Tram framework to orchestrate the saga, (4) enables monitoring and system-wide tracing, and (5) offers configurations for deterministic violation of service-level objectives (SLOs) in experiments. In addition to the architecture, we (6) provide defined realistic SLOs and load profiles to stress the architecture. This reference architecture aims to provide a benchmark solution for explaining self-adaptation and propagation of SLO violations across an architecture with complex patterns.

The remainder of this paper is structured as follows: First, we discuss related works in section II. Afterwards, we describe the T2-Project's architecture and implementation in section III and conclude in section IV.

II. Related Work

As related work, we examined other reference architectures for microservice patterns and self-adaptation. We looked at the TeaStore [9] and the SockShop [10] because they are still maintained to figure out whether any of them may be extended with the saga pattern, defined SLOs, configurable violations thereof, and self-adaptation.

978-1-6654-9493-9/22/$31.00 ©2022 IEEE
DOI 10.1109/ICSA-C54293.2022.00029

The TeaStore is a reference architecture for benchmarking and testing. As demonstrated by the authors, the TeaStore may be utilised to test self-adaptation technologies on the architecture. The architecture contains no self-adaptation of its own. Therefore, a developer who needs a self-adaptive system must invest additional effort to add the self-adaptation. The TeaStore comprises a few services responsible for the business logic, a service discovery service and a central persistence service, shared by all other services. All services are implemented with plain Java EE instead of a modern, more lightweight technology stack. A significant disadvantage is the synchronous communication via REST over HTTP APIs instead of providing lightweight asynchronous APIs. Regarding tracing and monitoring, the TeaStore is instrumented with Kieker [11]. Because of the TeaStore's lack of a modern microservice framework, e.g., Java Spring, the synchronous communication, and its central persistence , we decided against extending it. A significant drawback of the central persistence service is that the service's data is not distributed. Thus, there is no need for the saga pattern.

The SockShop is a reference architecture intended for testing and demonstration. It is implemented with microservice frameworks of different languages and implements the database-per-service pattern [5]. Similar to the TeaStore, most communications happen over synchronous REST calls. Furthermore, SockShop's services expose metrics, but it provides no out-of-the-box self-adaptation. We decided against extending the SockShop as it relies on synchronous API calls and provides no load profiles. Thus, it does not support the simulation of different load behaviours [9].

As for applications that explicitly implement the saga pattern, we found some example applications, such as the FoodToGo application[1]. However, as they are intended as examples for a particular topic, they are overly specific. Additionally, the documentation is lacking, increasing the difficulty of extending any of these applications to a broader scope.

III. T2-Project

This section describes the architecture of the T2-Project, the implemented patterns, especially the saga pattern, and the built-in self-adaptation. Furthermore, we outline the SLOs and how to configure the architecture to violate them.

During the remainder of this section, we refer to stakeholders using the T2-Project for benchmarking and testing as *the T2-Project's users*. In contrast, we refer to the imaginary customers who access the shop as *the shop's users*. The shop's users are simulated as requests from a load generator.

The T2-Project is open-source available on GitHub under the T2-Project Organization[2]. For a more realistic setup where each real-world microservice is independently developed [6], the project is organized in multiple repositories with one dedicated repository for each service.

[1]https://github.com/microservices-patterns/ftgo-application
[2]https://github.com/t2-project

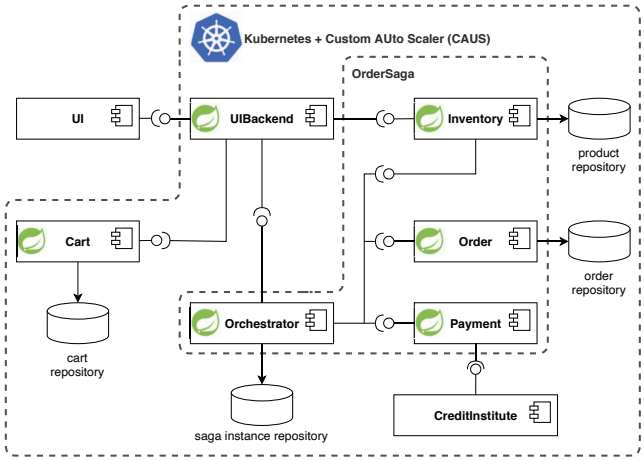

Figure 1. Component diagram of the T2-Project.

A. Architecture and Business Process

The T2-Project is loosely inspired by the TeaStore [9] and, therefore, simulates an e-commerce webshop that sells tea. The T2-Project's architecture is depicted in fig. 1 and consists of eight services which are described in the following.

The services follow the database-per-service pattern [5] concerning their domain data. All services that must communicate over REST over HTTP implement timeouts and retries. However, services that participate in the back-end's core business process communicate over lightweight and asynchronous messaging. The *UI* is the application's front-end where the shop's users would access the service to interact with the overall webshop. The front-end offers an overview of the available products, a shopping cart, the options to add units of a product to the user's shopping cart, delete them, and place an order containing the shopping cart's content. The *UIBackend* implements the API Gateway pattern [5] which accumulates responses from other services as per the *UI*'s request and relays the accumulated data to the *UI*. The *Cart* service is responsible for the shopping carts of the shop's users. The *Inventory* service manages the products offered by the webshop. As mentioned up front, the T2-Project implements the saga pattern using an orchestration instead of choreography, which allows a looser coupling of the services for improved future extensibility. While other example implementations of the saga pattern [5] often make one of the participating services the orchestrator of the entire saga, we decided to isolate the task of orchestration from the saga's participants according to the separation of concern and single responsibility principles which are crucial for microservices. Therefore, the *Orchestrator* service orchestrates the order sagas by publishing message events to the other relevant services. Thus, the *Orchestrator* service has no logic that directly contributes to the business goal. The *Order* service persists all orders to a database. It optimistically initializes the orders as success and, once they fail, marks them as failures. The *Payment* service contacts the shop's users' credit institutes to handle the money transfer. In

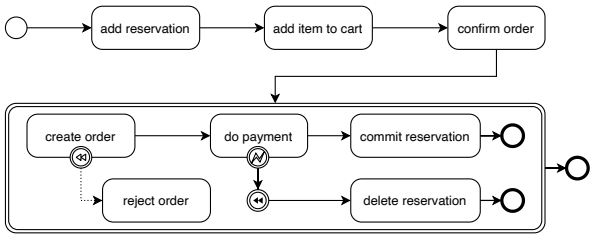

Figure 2. Business Process of the T2-Project in BPMN.

total, the *Inventory*, *Order* and *Payment* services participate in the order saga. The remaining service is the *CreditInstitute* service, which represents and simulates an external payment provider, such as PayPal, Klarna, or the user's actual credit institute. The *UI*, *UIBackend* and *CreditInstitute* communicate over synchronous REST APIs, and the saga participants communicate with asynchronous messaging through Apache Kafka. In addition, *Cart* and *Inventory* offer REST APIs to access their databases' content and *Creditinstitute* offers an REST API to configure the response behaviour at runtime. Furthermore, the architecture is deployable on Kubernetes and uses the Custom AUto Scaler (CAUS) [12] for ex-post explainable self-adaptations. Moreover, the services offer metrics for Prometheus and tracing spans for Zipkin.

The T2-Project's business process is depicted in BPMN in fig. 2. Furthermore, the mapping between the business process' tasks and the responsible service is shown in table I. According to the business process, the shop's users first select products and put units of them in their shopping cart. Next, they place an order, enter their payment details and confirm the order. Deleting items from the cart is possible as well but omitted for the sake of brevity. Once the shop's users confirm their orders, the orders are handed to the *Orchestrator*, which puts the saga participants into action. At first, the *Order* service creates a database entry for the new order. Afterwards, the *Payment* service contacts the *CreditInstitute*. If the payment is successful, the *Inventory* service updates the number of items in stock. However, if the payment fails, the *Inventory* service deletes all reservations associated with the failed order without updating the stock. Furthermore, the *Order* service must mark the order as "failed", thereby rejecting it. These steps must happen in a transactional context, i.e., either all of them or none are performed. Figure 2 denotes this with a transactional subprocess.

We evaluated the T2-Project's architecture and business process prior to the implementation by interviewing an experienced Software Architect and Azure AppConsultant at Microsoft. Based on his input, we improved the business process by adding reservations to the shop, i.e., for each item in a user's shopping cart, a reservation is placed at the inventory, thus, implementing a pessimistic lock. The reservations guarantee that after confirmation of the order, the order process cannot fail due to the webshop being out of stock of the requested products because of parallel purchases.

Service	Task
Inventory	add reservation, commit reservation, delete reservation
Order	create order, reject order
Payment	do payment
Cart	add item to cart

Table I
SERVICES AND THE TASKS THEY ARE RESPONSIBLE FOR.

Pattern	Involved / Implementing Service	Framework/Library
API Gateway [5]	UIBackend	-
Saga [5]	Orchestrator, Order, Inventory, Payment	Eventuate Tram Saga
Transactional Outboxing [5]	Orchestrator, Order, Inventory, Payment	Eventuate Tram
DB per Service [5]	Order, Inventory, Cart	-
Timeout [15]	Payment	Resilience4j
Processing Resource [16]	Orchestrator	-

Table II
PATTERNS AND THEIR LIBRARIES IMPLEMENTED BY THE T2-PROJECT.

B. Patterns and Implementation

The T2-Project's services are written in Java, using the Spring Boot framework[3] because it is popular, well documented, and provides many features that are relevant to microservice architectures [13]. Moreover, for implementing the saga pattern, the T2-Project utilises the Eventuate Tram framework[4]. Compared to other saga frameworks, Eventuate Tram has the least effort to use and the best execution [14]. As a standalone microservice framework, it is lacking. However, in combination with, e.g., Spring Boot, Eventuate Tram works well [13]. Table II displays an overview of the most relevant patterns used in the T2-Project. It notes which services implement it and which frameworks or libraries they use for each pattern. Missing entries in the *Framework/Library* column indicate that the service utilises no additional frameworks except the Spring Boot framework.

C. Service-Level Objectives and Self-Adaptation

The T2-Project's services provide data for monitoring and tracing. They use the micrometer instrumentation for Spring Boot to expose performance metrics compliant with Prometheus' format. For the distributed tracing, the T2-Project uses Spring-Sleuth to instrument its services. They provide tracing spans that comply with Zipkin's format. We decided to use Zipkin because a library exists that instruments the Eventuate Tram Framework for Zipkin. The traces are essential to ensure that the application runs as intended, and the metrics are crucial to check the SLOs and self-adapt the services accordingly. The T2-Project provides three predefined types of SLOs: (1) Availability SLOs, (2) Response time SLOs, and (3) CPU usage SLOs, e.g., used by CAUS [12] or the Kubernetes Horizontal Pod Autoscaler (HPA). In some cases, it might be impossible to ever adhere to the SLOs, e.g., because the cluster lacks computing capacity. If this happens,

[3]https://spring.io/
[4]https://github.com/eventuate-tram

the operator must adapt the thresholds. However, this is still less effort than defining the entire SLO from scratch.

Since the T2-Project's objective is to be used for analysing the impact of SLO violations, the services can be configured to violate their SLOs in a deterministic way, thus, enabling more straightforward experiments. Therefore, the *CreditInstitute* offers an interface to set the responses' delays to intentionally and deterministically violate its SLO resulting in the failure and rollback of the saga. Based on the SLOs, we add self-adaptation abilities to the T2-Project to handle changes in the environment [17]. SLO violations and self-adaptations may be triggered either by increasing load with the provided load profiles or by decreasing or increasing a service's response time via the provided interface.

The T2-Project's self-adaptation currently supports auto-scaling, realised with the HPA and self-healing within the limits of Kubernetes. The HPA's scaling policy considers the CPU utilisation SLO. All Pods have a set CPU and memory limit. The HPA is defined to scale out at a CPU utilisation of 70%. The other two SLOs are not yet covered for the autoscaling because the HPA relies on the pod metrics from the Kubernetes metrics server, which are restricted to CPU and memory utilisation. Furthermore, the HPA scales reactively after an SLO is violated. However, especially since the adaptations need some time until they take effect [15], a proactive autoscaler is needed to prevent service degradation. Because of its limitations, the HPA is only a temporary solution. Therefore, the desired solution is to use the CAUS [12], an autoscaler for containerised microservices on Kubernetes. However, CAUS was not adequately maintained during the past years and does not work on Kubernetes versions greater than $v1.21$. Nevertheless, we currently are implementing an improved version of CAUS as a Kubernetes Operator and plan to switch from HPA to CAUS.

To stress the T2-Project's services in experiments, we provide load profiles for Apache JMeter. Heavy workload results in the violation of SLOs and self-adaptation of services, which may either be used as input to novel tools, e.g., for impact analysis [18], or be compared to other auto-scalers.

IV. Conclusion, Benefits, and Future Work

This paper contributes with an open-source microservice reference architecture that implements complex microservice patterns, e.g., the saga pattern, and exports monitoring and tracing data for performance benchmarking. It comes with an out-of-the-box self-adaptation mechanism and defined SLOs to analyse the impact of SLO violations on a self-adaptive system. The architecture is implemented using state-of-the-art tools and technologies, for example, Java Spring Boot and Apache Kafka. Furthermore, the architecture can be deployed on Kubernetes alongside CAUS and the HPA for autoscaling. For tracing and monitoring, we instrument for Prometheus and Zipkin. Therefore, the T2-Project offers to software engineering researchers and practitioners (1) a more realistic benchmark solution for experiments with complex microservice patterns, (2) asynchronous and lightweight communication through messaging, (3) easier ex-post explainability of self-adaptation through CAUS, and (4) defined SLOs, configurable SLO violations, and load profiles.

For now, the T2-Project's self-adaptation capabilities encompass auto-scaling only, realised with the Kubernetes HPA, as CAUS does not work for the current Kubernetes version. However, we will replace the HPA with CAUS as soon we finish the latter. Furthermore, the predefined thresholds of our SLOs must be adapted for different infrastructure settings.

We plan to extend the project with additional patterns and other self-* properties in the future. Furthermore, we plan to provide an implementation of the T2-Project in other languages, e.g., NET.

References

[1] K. Baylov and A. Dimov, "Reference architecture for self-adaptive microservice systems," in *International Symposium on Intelligent and Distributed Computing*. Springer, 2017, pp. 297–303.

[2] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study." in *CLOSER (1)*, 2016, pp. 137–146.

[3] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*. IEEE, 2017, pp. 8–13.

[4] D. Taibi. A curated list of open source projects developed with a microservices architectural style. [Online]. Available: https://github.com/davidetaibi/Microservices_Project_List

[5] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[6] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. O'Reilly, 2021.

[7] G. Márquez and H. Astudillo, "Actual use of architectural patterns in microservices-based open source projects," 12 2018.

[8] J. Lewis and M. Fowler. (2014) Microservices: a definition of this new architectural term. [Online]. Available: https://martinfowler.com/articles/microservices.html

[9] J. von Kistowski *et al.*, "Teastore: A micro-service reference application for benchmarking, modeling and resource management research," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018, pp. 223–236.

[10] Weaveworks. Sock shop – a microservices demo application. [Online]. Available: https://github.com/microservices-demo

[11] A. Van Hoorn, J. Waller, and W. Hasselbring, "Kieker: A framework for application performance monitoring and dynamic software analysis," in *Proceedings of the 3rd ACM/SPEC international conference on performance engineering*, 2012, pp. 247–248.

[12] F. Klinaku, M. Frank, and S. Becker, "Caus: An elasticity controller for a containerized microservice," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: ACM, 2018, p. 93–98.

[13] I. Sailer *et al.*, "An evaluation of frameworks for microservices development," in *Advances in Service-Oriented and Cloud Computing*. Cham: Springer International Publishing, 2021, pp. 90–102.

[14] M. Štefanko, O. Chaloupka, B. Rossi, M. van Sinderen, and L. Maciaszek, "The saga pattern in a reactive microservices environment," in *Proc. 14th Int. Conf. Softw. Technologies (ICSOFT 2019)*. SciTePress Prague, Czech Republic, 2019, pp. 483–490.

[15] M. T. Nygard, *Release it!: design and deploy production-ready software*. Pragmatic Bookshelf, 2018.

[16] O. Zimmermann *et al.* Microservice api patterns. [Online]. Available: https://microservice-api-patterns.org/

[17] D. Weyns, "Software engineering of self-adaptive systems: an organised tour and future challenges," *Chapter in Handbook of Software Engineering*, 2017.

[18] S. Speth, "Semi-automated cross-component issue management and impact analysis," in *Proceedings of 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Nov. 2021, pp. 1090–1094.