FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Framework for the Choreography of Microservices using Mulesoft

João Castro Pinto

Mestrado em Engenharia Informática e Computação

Supervisor: Rui Filipe Lima Maranhão de Abreu

Second Supervisor: Bruno Miguel Carvalhido Lima

External Supervisor: Sérgio Brízida Lopes

July 28, 2023

A Framework for the Choreography of Microservices using Mulesoft

João Castro Pinto

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Jácome Cunha
Referee: Prof. João Saraiva
Member: Prof. Rui Maranhão

July 28, 2023

Abstract

This dissertation, developed in close collaboration with Deloitte, explores the practice of transaction management in the field of system integration. This field is responsible for designing solutions that enable the communication of systems of various natures. These solutions are typically built using platforms such as Mulesoft that focus on the creation of application networks via Application Programming Interfaces (APIs). The practice of the discipline enables agility in organizations, allowing faster responses to the needs of today's fast-changing digital world and delivering a competitive edge to enterprises.

The application of integration techniques often results in a distributed system. Distributed systems have, among other complexities, the need to carefully manage transactions. The Saga Pattern is a transaction management solution, however, its implementations are often tied to specific use cases, limiting their general applicability. This work will study the viability of a generic implementation of the Saga Pattern in Mulesoft using the choreography of microservices.

In this work, Shepherd, a Mulesoft framework, is proposed. Shepherd implements the Saga Pattern using microservices choreography in a generic manner, with the capability of adapting to different use cases. The framework uses graph-based representations of Sagas for the configuration of its application and its architecture is of distributed nature.

The validation of the framework is accomplished in two manners: by subjecting the framework's behavior through a set of tests, and by performing an experiment to check for efficiency improvements in the development of integrations. These validations revealed that Shepherd successfully implements the Saga Pattern and that a significant efficiency gain was caused by the usage of the framework in an asset reutilization activity.

This research concludes that the Shepherd framework has a high potential for applicability in the industry and, in a way, challenges common approaches to integration development such as API-led Connectivity. The usage of Shepherd in the integration of systems introduces an API whose business logic is configurable, which is a novel concept in the field.

Acknowledgements

Above all, I thank both my parents for having raised and educated me. I am profoundly grateful for the opportunities and experiences they provided me during my education. I extend this sense of gratitude to my siblings and to the people in my family who were tirelessly driven to promote education for their descendants and relatives.

Furthermore, I thank Deloitte for having me while I developed this work. I learned tremendously in these last months. I especially express my appreciation to Sérgio Lopes, my supervisor, for all the insightful and productive conversations we had during the development of this work. Along the journey, Bruno Vasquez and Paulo Branco also joined our conversations to share their valuable knowledge, to whom I express my deep gratitude.

Lastly, I thank my supervisors Professor Rui Filipe Lima Maranhão de Abreu and Professor Bruno Miguel Carvalhido Lima for their thoughtful advice and questions from the beginning until the end of this work. My supervisors always posed the hard questions, whose sole intent was to contribute to the success of this dissertation, and my growth as a producer of scientific work.

João Castro Pinto

"Que beleza é conhecer o desencanto
Viver tudo bem mais claro no escuro"

Tim Maia

Contents

List of Figures

Listings

Abbreviations and Symbols

ACID	Atomic, Consistent, Isolated, and Durable (Transactions)
API	Application Programming Interface
ERP	Enterprise Resource Planning
gRPC	Google Remote Procedure Call
HTTP	Hypertext Transfer Protocol
IT	Information & Technology
PoC	Proof of Concept
REST	Representational State Transfer
SOA	Service Oriented Architecture

Chapter 1

Introduction

1.1 Context

This dissertation was done in collaboration with Deloitte. Deloitte is recognized as one of the world's largest professional services companies, both in terms of income and the size of its professional workforce [1]. The firm is part of the prestige "Big Four" auditing and consulting entities. Deloitte's activity extends to many sectors, with a significant presence in the field of systems integration.

This area is responsible for creating strategies and solutions that allow different systems, applications, and Information & Technology (IT) environments to work together, delivering more efficient operations. As businesses increasingly adopt digital technology, the role of integration becomes crucial, providing agility to organizations, which is essential in today's fast-paced world.

Deloitte's excellence in the field of IT systems integration hasn't gone unnoticed, particularly in its utilization of MuleSoft, a leading platform in the field. MuleSoft has recognized Deloitte's achievements with multiple awards [2]. These honors corroborate Deloitte's proficiency in employing MuleSoft for developing robust, flexible, and scalable integrations. The awards serve as an industry endorsement of Deloitte's leadership in the space and its ability to leverage MuleSoft's potential in addressing complex integration challenges.

Deloitte has proposed this thesis as a solution to the absence of possibilities to generically implement the Saga Pattern in MuleSoft. It is a step to further enrich Deloitte's mastery of systems integration. The company wishes to explore new methodologies and frameworks that can potentially optimize the development of integrations by making it more efficient. This research explores how effectively the Saga Pattern can be implemented in a distributed system and measure its impact on development. It is within the context of striving for continuous improvement that this opportunity of research unfolded, aiming to provide valuable insights that can be integrated into Deloitte's existing body of knowledge and practices.

1.2 Problem Statement

In the realm of software development, the rise of microservices architecture has brought forth the challenge of managing distributed transactions across multiple services. This issue is particularly pronounced in complex applications where multiple independent services must collaborate to complete a transaction. Traditional distributed transaction techniques often are not suited, as they can lead to resource locking and impose a lack of autonomy on individual services.

The Saga Pattern [3] has emerged as a solution to managing distributed transactions in microservices. However, the implementation of the Saga Pattern often requires careful design and coordination among services, making it a complex task. Moreover, existing implementations of the Saga Pattern are often tied to specific use cases, limiting their general applicability [4].

In this context, the need arises for a solution that can implement the Saga Pattern in a generic form, adaptable to different use cases. This is where the Shepherd framework comes in. Designed to work with MuleSoft, an integration platform mainly for building application networks, Shepherd provides a way to implement the Saga Pattern in a generic form using the choreography of microservices. It can be integrated into an API-led environment or not, depending on the use case, offering a flexible solution to the challenge of managing distributed transactions in a distributed environment.

1.3 Objectives

The purpose of this work is to address the need for a generic implementation of the Saga Pattern, which is adaptable to different use cases. Thus, this dissertation aims at studying the viability of implementing the Saga Pattern using the choreography of microservices in MuleSoft in a generic manner.

The specific research objectives behind the creation of the framework are the following:

- Propose a MuleSoft framework that generically implements the Saga pattern using choreography of microservices;
  - The framework's user experience should be developed with a configuration-first approach;
- Accelerate the process of reutilization of APIs in an API-led environment;
  - The impacts on the API-led strategy of applying such a framework should be studied and analyzed.

1.4 Document structure

Besides the present chapter, which introduces the dissertation, the present document contains four more chapters. The next chapter provides background information on digital transformation and

integration, microservices and introduces the concept of the Saga Pattern for managing transactions in microservices. Also, it sets forth tools that are similar to the proposed framework. The third chapter presents "Shepherd": the proposed solution for managing transactions in a distributed environment. It provides a high-level description of Shepherd, its representation of Sagas, and its architecture. Additionally, the fourth chapter presents the validation of Shepherd, including the setup, results, and discussion of the validation process. The fifth and last chapter concludes the document, summarizing the findings and suggesting future work. It also includes final statements about the study.

Chapter 2

Background & State of the Art

2.1 Digital Transformation & Integration

As Vial [5] puts it, digital transformation is "a process that aims to improve an entity by triggering significant changes to its properties through combinations of information, computing, communication, and connectivity technologies." Ultimately, the goal of digitally transforming an organization is to increase its productivity and efficiency [6]. As it is well known, it is very difficult for businesses that do not adhere to the current digital trends to stay competitive [7]. The pandemic emphasized this observation even more, as digital means of communication was perceived as the only *safe* way to communicate, at least for some time [8].

The responsibility for the digital transformation of an organization mainly lies within the IT department [6]. The IT teams should be able to address the company's needs, from the digitalization of business processes to agilizing communications in the organization. As stated by van Gils et al.[9], a simple business alignment with the IT department is not sufficient: the difference between business and IT is fading every day. Thus, the digital transformation of a company requires the insights of the organization's management team, the IT department, and any other internal stakeholders.

Integration is the act of composing a system by joining "two or more things to create a whole" [10]. In the context of digital transformation, integration is leveraging digital integration techniques, such as exposing enterprise assets and services through Application Programming Interfaces (APIs) and governing them via API managers. This approach allows organizations to "seamlessly integrate, unify and standardize core business capabilities across diverse IT environments" [11]. In other words, integration is responsible for creating links between systems that normally would not communicate in order to create a fulfilling and comprehensive IT infrastructure that addresses an organization's business needs. It is a critical aspect of digital transformation. Integration's purpose is to increase an organization's performance and its ability to respond to the rapidly evolving business needs of today's digital world.

2.2 Microservices

2.2.1 What are Microservices?

The microservices architecture is an approach to applications development by creating several small, context-bounded services which collaborate with each other towards a shared goal. These services have a cohesive, (ideally) small set of responsibilities and their software lifecycles are independent of each other. Each service should have a well-defined, stable interface, and, typically, the communication between them is done with standard lightweight communication protocols such as Hypertext Transfer Protocol (HTTP) or Google Remote Procedure Call (gRPC) [12].

This approach to developing applications has arisen as a next step from the Service Oriented Architecture (SOA) [13], and more generally, monolith architectures. This happened because of the scaling requirements developers would face with the increasing demand for their software products [12]. It turned out monoliths do not scale well. Large monoliths have huge codebases, which are very hard to maintain and understand [14]. The initially well-designed component modularity erodes over time and good software architecture practices, such as the low coupling and high cohesion of software components often do not prevail. This leads to a decrease in developers' productivity and difficulties in staying competitive in today's markets. Since a monolith is composed of one single application, it is only possible to deploy the whole application to production, which, when dealing with a large monolith, will not go well every time and may jeopardize the business goals of an organization [15].

The microservices architecture is a solution for these problems. By splitting a monolith into a set of microservices, i.e., a set of applications, one can independently develop, test and deploy the different services, that, together, would compose the monolith [15]. Because the microservices are split from each other and have a specific set of responsibilities, it is much harder to break the initial thought-out modularity and good practices of software architecture. Furthermore, the codebase of each service is significantly smaller, which leads to easier comprehension and maintenance of the application [12].

Since there are no silver bullets in software architecture, the implementation of the microservices architecture also poses significant challenges [15]. By functionally splitting an application, typically, each service has a database of its own [16]. A microservice is responsible for the data related to its context and should store it, maintain it, and provide it when necessary. What was a simple ACID (atomic, consistent, isolated, and durable) transaction in a monolith must now be distributed among several applications. Some form of distributed transaction management system must take place when using microservices [12]. Also, performing complex data queries becomes hard to manage, as data needs to be gathered from several points and probably manipulated. Besides data management, testing the whole system becomes more complex as the functionality of the system is distributed [14]. Furthermore, developers must now deploy several artifacts instead of one single artifact, which (again) increases complexity. Lastly, according to Conway's Law, if the implementation of a microservices architecture is to succeed, the structure of developer teams must adapt to correspond to the communication of services [12].

2.2.2 Microservices Collaboration Patterns

As it was stated, microservices are described as modular and independent units of software, "organized around business capabilities" [13]. In order to form complex applications and fulfill business needs, microservices collaborate towards a shared goal. In a monolith, the collaboration between software modules is done through "a simple language-level function call" [13], whereas, in a microservices architecture, network-level communication is required.

A microservices collaboration pattern defines how this network-level communication between services is done. It establishes a framework for microservices communication which will be the base for fulfilling business requirements of complex applications [13]. Responding to business needs involves executing application processes, which are a sequence of tasks (potentially) executed by different microservices. Some frameworks in the market, such as Netflix Conductor [17] and Uber Cadence [18], which will be described further in Section 2.4, also refer to application processes as workflows.

In the literature [12, 13, 19, 20, 21], two collaboration patterns are widely discussed: orchestration and choreography. In real-world scenarios, a hybrid version of these patterns is used [18, 17]. The following sections will present these patterns.

2.2.2.1 Orchestration

Orchestration is a collaboration pattern that relies on a central service to orchestrate several tasks to execute an application process [12]. This central point of orchestration is itself a microservice and performs requests to other microservices to execute their tasks within an application process [22]. In Figure 2.1, "Order Service" is the orchestrator and requests other microservices to execute their part of the job.

In this context, microservices fall into two categories: composite or atomic [13]. Composite microservices are orchestrators, i.e., microservices that invoke other microservices in order to deliver their task. Atomic microservices are the ones that do not rely on other microservices: their responsibilities are local to their system. An interesting remark is that composite microservices can invoke other composite microservice to compose more complex application processes.

Besides being an intuitive collaboration pattern [13], there are other benefits to using orchestration. Service orchestrating from a central controller enables easy tracking of the execution of the application process, which eases debugging and understanding the system [20]. Furthermore, since a format of request-reply is used, it is possible to predict response times, even when failures occur [13]. That can be achieved with timeouts and a maximum number of retries in case of failures. When interfacing with an end-user, predictable response times are essential in order to guarantee user satisfaction [23].

The main trade-off that should be considered when using orchestration is having a single point of failure in the system. The orchestrator's implementation must be highly available, fault-tolerant, and scalable. For this reason, introducing an orchestrator to a microservices architecture is a complex task and requires careful planning and robust design strategies. It also necessitates

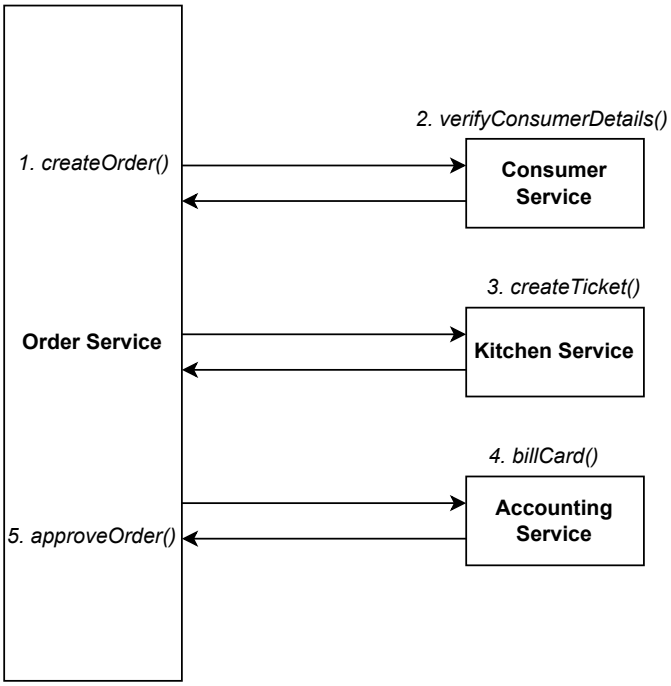


Figure 2.1: A high-level design of microservices orchestration to create orders in a food delivery service.

thorough testing to ensure that the orchestrator can handle different failure scenarios and high-load conditions without compromising the overall performance and reliability of the system.

2.2.2.2 Choreography

Choreography is a microservices collaboration pattern that follows an event-driven approach [12]. In this pattern, each microservice is designed to respond to events it receives and to generate new events as needed. It allows for application and business processes to be defined by a sequence of events, whose responses implement the processes' business logic. It is an alternative form of connecting different software components to work towards a shared goal. Figure 2.2 presents a high-level design of a service choreography.

A remarkable characteristic of pure choreography is the absence of a central controller [22]. This is its main difference to the orchestration pattern. Following an event-driven approach, there is no need to have a central orchestrator to invoke the different microservices: raising events in a microservice triggers the execution of tasks in other microservices.

However, this approach creates the need for asynchronous and persistent communication channels, such as event brokers. It is critical to guarantee the persistence of events in the case of failures [12]. Also, event brokers abstract callers from communicating with another microservice; it sends an event to a queue that is highly available [15]. Without event brokers between microservices, a robust choreography becomes unfeasible.

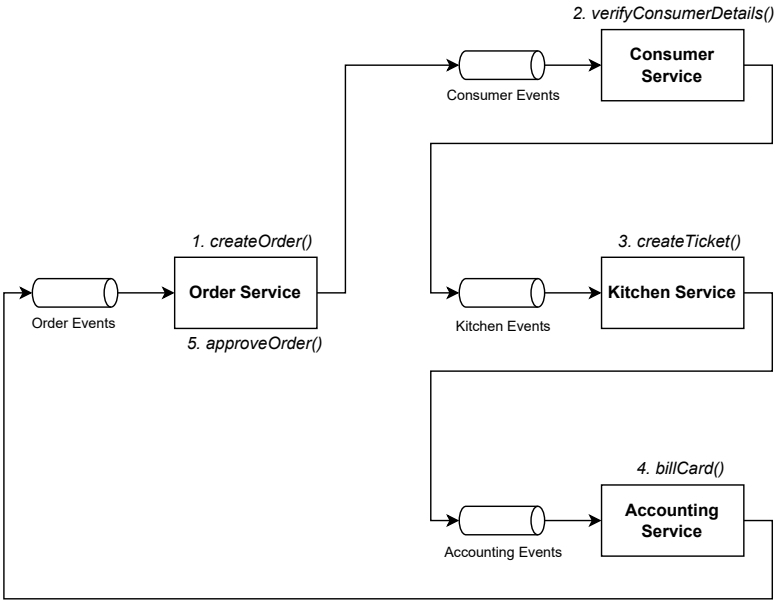


Figure 2.2: A high-level design of microservices choreography to create orders in a food delivery service.

There are several benefits to using choreography as a collaboration pattern of microservices. As it is an event-driven pattern, it allows for faster processing of tasks, since it opens the possibility of executing tasks in a parallel fashion without depending on a central controller service [19]. As there is no central point of control, in pure choreography, there is no single point of failure of the system, which tends to increase the system's robustness. Furthermore, it contributes to the loose coupling of microservices [13]. This is possible because event brokers must guarantee that events are not lost by using persistent queues [15]. Another benefit of using the choreography pattern is that it promotes low chattiness [13]. Events are published only when a task is completed or if there is a state change; there are no reply messages to flood the network.

Just like any solution in engineering, some trade-offs should be taken into consideration when applying the choreography collaboration pattern. As with any purely event-driven approach, one of the biggest challenges is monitoring and understanding the flow of execution at runtime [19]. In pure choreography, where the pattern is applied to a specific use case, complex application processes which involve a large number of microservices may lead to a 'spaghetti' architecture as connections between microservices are made point-to-point [13]. Furthermore, it is important to consider that the design of a choreographic system is quite complex, as it necessarily involves more components, such as event brokers [19]. Lastly, application processes invoked through choreography have an indeterminate response time [13]. Each microservice is called in a fire-and-forget model, which makes it challenging to predict a response time of an application process.

2.2.2.3 Hybrid Pattern

According to Megargel et al. [13], the hybrid collaboration pattern is a common ground between the orchestration and choreography patterns. It leverages the asynchronicity of choreography, as well as the process visibility of orchestration. It has a central point of control, the orchestrator, just like the orchestration pattern. However, the collaboration itself is done in a similar way as in the choreography pattern. Microservices communicate via events; in order to invoke another microservice, a microservice needs to publish an event to the orchestrator. Of course, it depends on the implementation, but typically, the orchestrator then routes the events to the respective subscribed microservice. As in choreography, upon receiving the event, the microservice executes the required activity, which is part of an application process. In Section 2.4, two pre-built open-source process engines that implement the hybrid pattern will be presented.

This pattern leverages the benefits of both patterns. From the choreography side, it inherits the possibility of parallel processing and promotes lower chattiness than pure orchestration. Furthermore, from orchestration, it leverages process visibility, which is essential for debugging and monitoring executions. Also, since all events flow through the central orchestrator, it is easy to ensure predictable response times to outside clients.

The main trade-offs of this option are three-fold: firstly, just like the orchestration pattern, it has a single point of failure. In addition to that, a process engine is a complex program, thus its implementation is not trivial.

2.2.3 Transactions in Microservices: the Saga Pattern

One of the primary challenges of microservices architecture is that it doesn't support traditional ACID transactions due to its distributed architecture [12]. ACID properties of transactions guarantee the reliable execution of database transactions [24]. In a microservices system, where multiple services may be involved in a single transaction, maintaining the ACID properties becomes a challenge [15]. As a result, developers need to look for alternative solutions that can provide similar guarantees.

The traditional way of performing transactions in a distributed system is using distributed transactions, like, for example, 2-phase commit. These types of transactions offer the ACID properties in transactions across several applications, however, their application in a microservices architecture poses several challenges. Firstly, as Richardson [12] presents, distributed transactions like 2-phase commit are not supported by modern technologies, such as MongoDB[1], Cassandra[2], RabbitMQ[3], and Apache Kafka[4]. Thus, when using 2-phase commit, developers and architects must refrain from using a significant set of modern technologies. Furthermore, and most importantly, distributed transactions restrict system availability. For a distributed transaction to be committed, all of its participants must be available. This significantly decreases the availability

[1] https://www.mongodb.com/
[2] https://cassandra.apache.org/
[3] https://www.rabbitmq.com/
[4] https://kafka.apache.org/

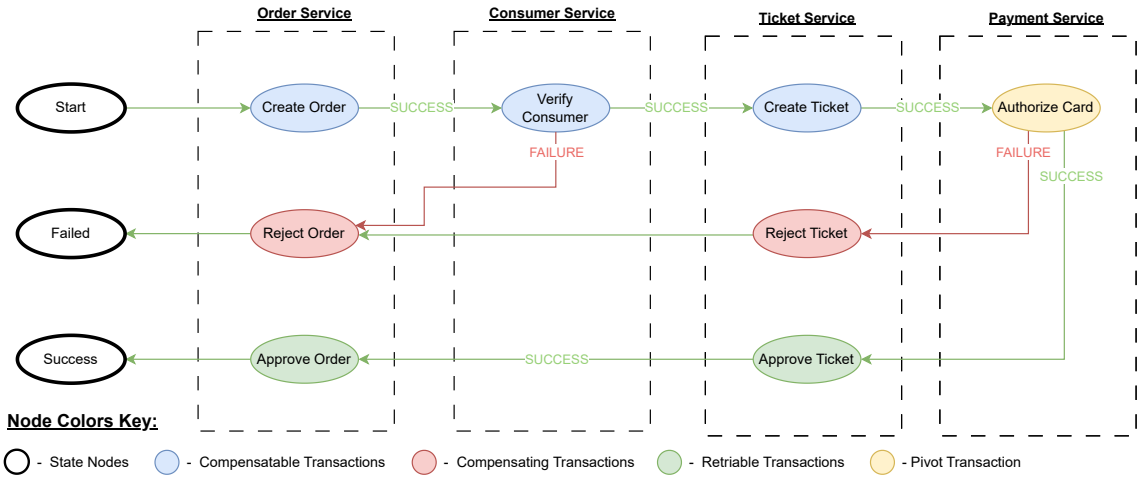


Figure 2.3: Representation of a Saga to create orders in a food delivery system.

of the system, which is not desired. This is a trade-off as dictated by Eric Brewer's CAP theorem [25]: the choice between availability and consistency.

As it was initially presented in [3], one approach to transaction management in a distributed architecture is the Saga Pattern. The Saga Pattern provides a way to manage transactions in a distributed architecture with several databases. The Saga Pattern uses a sequence of local transactions to perform the equivalent of what would be a single transaction in a monolith application. The main principle behind Sagas is that the completion of a local transaction will trigger the execution of the next local transaction. Based on this principle, Sagas can be built by a complex sequence of transactions, creating long and parallel chains of execution of local transactions. Observe that parallel Sagas can be created when a transaction triggers the execution of a set of transactions. By doing so, the parallel execution of transactions will occur in the system.

As it is put by Richardson [12], a Saga is made up of a series of local transactions, each of which is responsible for updating the state of a single service. It is assumed that the local transactions have ACID properties. There are four types of transactions in a Saga: compensatable, pivot, retriable, and compensating transactions. Compensatable transactions are steps of the Saga that can fail while performing changes. Each compensatable transaction may have a compensating transaction associated that rolls back the changes made by the compensatable transaction in the event of a failure [26]. Observe that some compensatable transactions do not have compensating transactions, or do not fail[5]. The pivot transaction is, in a sense, the last compensatable transaction of a Saga and the first retriable transaction. If it succeeds, the whole Saga will run until completion. Retriable transactions are transactions that confirm the changes made by the Saga and that always succeed, provided their execution has no timeout constraints. Compensating transactions also hold this property. Furthermore, note that because of the ACID properties of local transactions, whenever a transaction fails, it can be assumed that the changes it performed were rolled back.

[5]It will be explained further in the section why it can be assumed that some transactions do not fail.

Figure 2.3 represents graphically the structure of a Saga. Firstly, the sequence of compensatable transactions is executed. If any of those fail, the changes made by the previously successful compensatable transaction will be rolled back by the execution of the respective compensating transactions. When all changes have been rolled back, the Saga execution finishes in a failed state. However, if the compensatable transactions are successful, the pivot transaction is executed. The pivot transaction is the last transaction to decide if the Saga fails or not. If it succeeds, the retriable transactions will be executed. Since these transactions do not fail, the Saga will terminate with success. On the other hand, if the pivot transaction fails, all the changes made by compensatable transactions will be rolled back by compensating transactions.

Sagas can be implemented with choreography or orchestration. The selected collaboration pattern constitutes a layer of abstraction upon which the Saga Pattern is implemented. This concept is very well explained by [27]. According to Friedrichsen [27], the Saga Pattern is implemented on top of a technical layer that deals with technical issues, such as the availability of Saga participants or errors internal to the participants. Following this rationale, the Saga pattern is implemented in a higher layer which is abstracted from the technical issues that the lower layer might have to deal with. The purpose of having two layers is to segregate the responsibilities of dealing with technical and business errors.

As Friedrichsen [27] puts it, a "business error occurs if an existing business rule is violated. E.g., if you try to pay with an expired credit card or if a pending payment would exceed your credit limit: That is a business error. From a technical point of view, everything is fine." On the other hand, a technical error happens when the technology or infrastructure fails, i.e., a system is unavailable, or the execution of a local transaction fails. A business error should be handled by the Saga layer by executing compensating transactions, and a technical error should be handled by the lower layer, where the collaboration pattern is implemented.

Handling technical errors is more challenging than handling business errors [27]. Asynchronous and persistent event brokers allow tasks to be retried indefinitely until their success. However, there may be times when the error will persist forever, which would impede the Saga layer from completing the Saga, leaving the system in an inconsistent state as some transactions of the Saga would not be executed. Compensating transactions do not solve this problem, as this situation may occur when executing any type of transaction. A naive solution would be to compensate compensating transactions that have technical problems, but that would just create the same situation with the new compensating transaction. In these cases, escalation strategies must be in place. Human intervention or techniques such as event sourcing [12] can reset the system to a valid state in case of a non-recoverable failure. Obviously, the failure needs to be addressed when the system is recovered so that it (ideally) does not happen again.

The reliance of the Saga Pattern on the technical layer is the reason why it can be assumed that retriable, compensating, and some compensatable transactions always succeed in a Saga. As the Saga layer is solely responsible for handling business-level errors and in these types of transactions business errors do not occur by definition, it can be assumed that retriable and compensating transactions always succeed. However, since any computer is subject to technical issues, a robust

technical layer must be implemented below the Saga Pattern.

2.3 MuleSoft

MuleSoft[6] is a platform that provides a set of tools for integration. With MuleSoft, it is possible to integrate "data and systems" and automate "workflows and processes". Its purpose is to enable IT teams to create the "digital building blocks that teams can use as they need", while considering the necessary "security, governance, and compliance" aspects [28]. MuleSoft adopts a composable approach to integration: its goal is to convert digital assets into reusable products. In order to do so, MuleSoft follows an API-led Connectivity approach.

The main product of MuleSoft is the Anypoint Platform. Although the platform offers several services, the main ones are the Design Center, Anypoint Studio, Anypoint Exchange and the Management Center. A description of each component follows. In simple terms, the Design Center is a code editor oriented for the development of API specifications [29]. Typically, the API specification language that is used in Design Center is RAML, one of many API specifications languages that formally establish API contracts. Anypoint Studio is an IDE dedicated to the development of the applications which serve underneath the abstraction layer that APIs provide, using a low-code/no-code approach [30]. It features a large set of pre-installed connectors that can be used to interface with different systems and integrate them. Anypoint Exchange is a marketplace where reusable digital assets are published and made available to be used by others [31]. It also provides useful features such as versioning, mocking services and API cataloging. Lastly, the Management Center is used for API governance and management, and also for deployment, both in MuleSoft's cloud infrastructure - CloudHub - and on-premises deployment [32].

As it is discussed in [33], when using MuleSoft, the development of APIs for integrating data and services is reliant on all the different products described above. Because MuleSoft defends an API-first approach, the development of an integration application usually starts in the Design Center, where the API specification is created. With the API specification, Anypoint Studio is used for developing the application that is running behind the API and that will serve the requests made through the API. In this stage, assets from Anypoint Exchange may be used and integrated into the application. When the development of the application in Anypoint Studio is complete, the application may be deployed using the Management Center. In order to allow other applications to easily interface with the deployed application, a connector for the deployed application may be published to Anypoint Exchange. Besides the connector, code fragments, and policies (among others) that were developed for the deployed application may be published to Anypoint Exchange to be reused as well.

[6] https://www.mulesoft.com/

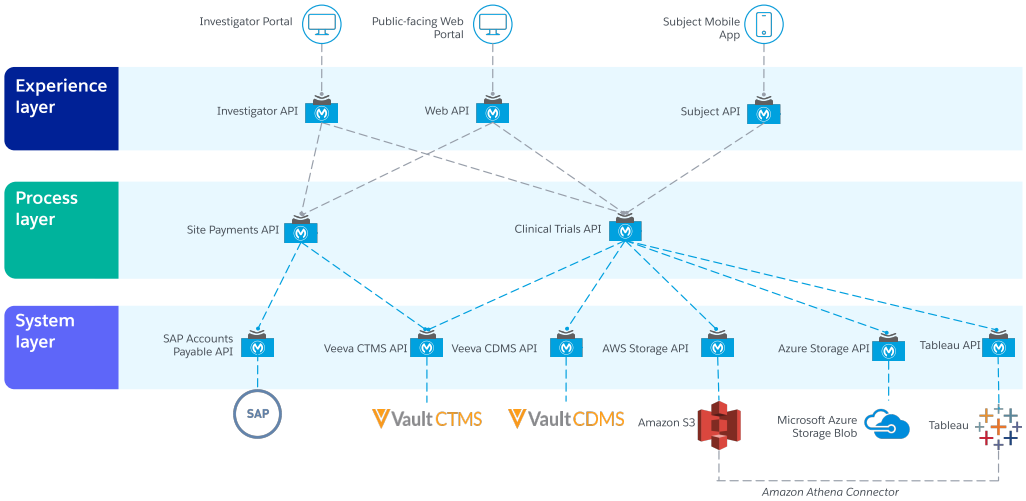


Figure 2.4: Example of the application of the API-led Connectivity approach. [36]

2.3.1 API-led Connectivity

As Sharma et al. [34] states, API-led Connectivity is an approach to building and managing integrations between different systems, applications, and services using APIs. It involves designing and implementing APIs as the primary means of connecting and integrating various components of a software ecosystem, enabling them to communicate and share data in a standardized and efficient manner.

As represented by the example in Figure 2.4, API-led Connectivity generally involves three layers of APIs: the system, process, and experience layers. Typically, in the context of this approach, APIs only communicate to APIs that are on a lower or the same level of abstraction [35]. Requests originating from external systems interface with experience APIs [35]. The following is a description of each layer, and the layers are ordered ascendingly by the level of abstraction.

1. **System Layer**: This layer involves exposing APIs that provide access to the core systems and databases of an organization. These APIs allow other systems or applications to interact with backend systems, such as databases, legacy systems, or third-party systems, in a controlled and secure manner [37].

2. **Process Layer**: This layer exposes APIs that encapsulate specific business processes or services, such as order processing, payment processing, or customer management. These APIs abstract the underlying systems and provide a higher level of abstraction, making it easier to build composite applications and workflows [37].

3. **Experience Layer**: This layer involves exposing APIs that provide a consistent and unified experience to end-users, typically through channels such as web, mobile, or IoT devices.

> These APIs enable developers to create user interfaces and experiences that are optimized for different devices and platforms while leveraging the same underlying business logic and services [37].

As Choudhary [37] affirms, API-led Connectivity allows organizations to adopt a modular and agile approach to integration, which leads to looser coupling between services. Furthermore, it also enables organizations to expose their internal capabilities as APIs, opening up new opportunities for innovation, collaboration, and monetization. API-led Connectivity is often used in modern application development approaches, such as microservices architecture, where APIs are the primary means of communication between loosely coupled and independently deployable services.

The idea of having three separate layers and several APIs in order to address business needs is to eliminate point-to-point connections. This approach leads to a more maintainable, manageable, and reusable codebase. As stated before, APIs in the top layers are utilizing the APIs in lower layers. The main benefit of the API-led approach is that if a business need for a new API arises, its implementation will be facilitated by the APIs in the lower layers. This allows for an agile response to the rapidly changing environment that is the IT world, in a sustainable manner.

API reutilization is one of the cornerstones of API-led connectivity. The reuse of APIs allows organizations to accelerate the development process as they do not need to repeatedly build the same interfaces or connections for different applications. This not only saves time and resources but also ensures consistency across different applications, leading to less complexity and easier maintenance. Moreover, API reutilization fosters innovation as developers can leverage existing APIs to create new services and functionalities. From a broader perspective, API reutilization aligns with the principles of modularity and loose coupling, making systems more flexible, scalable, and resilient to changes.

2.3.2 Mule Apps & Mule Runtime

As it is explained in [38], Mule Runtime is the runtime environment for executing Mule applications, which are integration solutions built using MuleSoft's Anypoint Studio. Mule Runtime provides the core runtime capabilities for running Mule applications, including connectivity, messaging, transformation, orchestration, error handling, security, and scalability. It is a Java-based runtime engine that hosts and provides the necessary runtime infrastructure for integrating applications, data, and devices in a unified and efficient manner.

The difference between a Mule Application and an API is the following: a Mule Application runs below the layer of abstraction that APIs provide. The Mule Application is responsible for fulfilling the contract that defines the API, i.e., it is in charge of providing the services its API claims to provide [39]. The concepts of APIs and applications are often used interchangeably, however, it is important to be able to differentiate between the two concepts.

2.3.3 Microservices in MuleSoft

In MuleSoft, the implementation of a microservices architecture is accomplished through the use of Mule Apps and APIs. As previously discussed in Section 2.2, a microservice is essentially a small application responsible for a well-defined business domain, typically exposed through a communication protocol such as a Representational State Transfer (REST) API. MuleSoft's platform is designed for creating such applications and exposing them through APIs, which are then integrated into an application network to meet business needs. This approach aligns closely with the principles of a microservices architecture, making MuleSoft a viable option for its implementation.

The concepts of API-led Connectivity and the microservices architecture often complement each other. However, it's important to distinguish between the two. The microservices architecture is an architectural pattern that decomposes complex applications into small, loosely coupled, independently deployable, and scalable services. In the context of this work, the terms API, microservice, and service are used interchangeably to reflect this pattern. API-led Connectivity is an approach to building integrated systems through the use of APIs in a standardized an efficient manner.

Microservices communicate with each other over APIs, which could be REST HTTP APIs, gRPC APIs, or other technologies. These APIs can be designed and implemented using the principles of API-led Connectivity, further reinforcing the complementary nature of these concepts.

https://www.overleaf.com/project/641824f686b904e759c1d0aa

2.4 Similar Work

In this section, two workflow orchestration engines will be briefly discussed that may be applied to similar use cases as the framework proposed by this work. However, the following or other similar tools are not compatible with MuleSoft's platform, a problem that this work aims to solve.

From the several workflow orchestration engines available, Uber Cadence [18] and Netflix Conductor [17] were chosen to be discussed here, as they were used as an example by the author's Deloitte supervisor to describe the goals of this work. Additionally, these are among the most popular workflow orchestration engines [40].

2.4.1 Netflix Conductor

According to its documentation [17], Netflix Conductor is a cutting-edge, open-source workflow orchestration system that provides a scalable and reliable solution for managing complex workflows in large-scale, distributed systems. At its core, Conductor is designed to simplify the management of complex workflows in a distributed environment. It allows users to define, schedule, and execute workflows as a series of tasks, each represented as a microservice. Workflows in Conductor are defined using a declarative JSON-based language, making it easy to specify the sequence of tasks, their dependencies, and any input/output data. Conductor also provides powerful

tools for monitoring and managing workflows, including a web-based user interface and extensive APIs for programmatically interacting with the system.

One of the key features of Netflix Conductor is its scalability. It is designed to handle high volumes of workflow tasks, making it suitable for large-scale applications with millions of concurrent workflows. Conductor utilizes Dynomite [41] as a storage system, and uses HTTP as the communication protocol to communicate with other nodes. It also supports horizontal scaling by allowing multiple instances of Conductor servers to work together in a cluster, providing fault tolerance and high availability. From a functional point of view, Conductor supports a wide range of workflow patterns, including sequential, parallel, and conditional branching, making it suitable for diverse use cases. It also provides advanced features, such as dynamic task routing, event-driven triggers, and custom workflow extensions, which allow users to extend the functionality of Conductor to suit their specific requirements.

2.4.2 Uber Cadence

According to its documentation [18], Uber Cadence is a powerful, open-source workflow orchestration system developed by Uber, a leading global ride-hailing and logistics platform. Cadence is designed to provide a reliable and scalable solution for managing complex workflows in distributed systems, particularly in scenarios where business processes require coordination across multiple microservices.

One of the notable features of Uber Cadence is its fault-tolerant and scalable architecture. Cadence uses a distributed, multi-master architecture that provides high availability and fault tolerance. It uses a combination of a centralized metadata service and distributed storage systems, such as Apache Cassandra, for managing workflow state and history. This way, Cadence provides horizontal scalability by allowing multiple Cadence servers to be deployed in a cluster, enabling efficient handling of high volumes of workflow tasks. Regarding functionality, just like Conductor, Cadence allows users to define workflows using familiar programming constructs, such as loops, conditionals, and error handling. This makes it easy for developers to express complex business logic in workflows and customize the behavior of workflows to suit specific requirements.

2.5 Research Gap

The research gap this dissertation aims to fill is the study of the viability of leveraging MuleSoft's platform to implement the Saga pattern using the choreography of microservices in a generic form. MuleSoft, as it was discussed, is typically used in the context of an API-led Connectivity approach. As of today, there is no standardized manner of implementing the Saga pattern with MuleSoft. The purpose of this work is to build a framework that will serve as an accelerator for the development of these systems and validate whether it has potential applicability in the industry.

Chapter 3

Shepherd

This chapter presents Shepherd: a MuleSoft framework for implementing the Saga Pattern using the choreography of microservices. This chapter is composed of five parts. It starts by describing the framework from a high-level perspective. Afterward, the chapter explains how Sagas are defined by the Shepherd framework, followed by an explanation of its main functional mechanisms. Additionally, a description of the framework's architecture is present, and lastly, a shared responsibility model is presented that explains the principles behind the framework's usage.

The name Shepherd captures the essence of this framework by drawing an analogy to agriculture. In a similar way that a shepherd guides and manages the activities of a group of mules, this framework will establish how a set of Mule applications collaborate towards a shared goal by performing a set of managed activities.

3.1 High-level description

The purpose of this framework is to implement the Saga pattern using MuleSoft in a generic manner, independently of the context. As it was described in the Section 2.2.3, the Saga pattern can be implemented using different collaboration patterns, and in this work, microservices choreography will be used. Shepherd will abstract its user from the complexities of implementing the Saga pattern, requiring the user only to specify how the Saga is composed by defining the transactions and precedence relations between them. In doing so, the user will be indicating business-related recovery scenarios by defining compensating transactions. Besides that, it will also be the responsibility of the user to create and deploy the microservices, which will be the Saga participants. This also implies implementing the business logic of the transactions.[1]

Having defined the Saga, created and deployed the microservices, and implemented the transactions which compose the Saga, Saga execution requests can be performed to an API. This API will be referred to as the Interface. The Interface will then trigger the execution of the first transaction of the requested Saga, which, upon its completion, triggers its subsequent tasks. When the execution of a Saga is triggered, an execution context will be created, which will store the state

[1] The terms "task" and "transaction" will be used interchangeably from this point onwards.

of the Saga and keep track of the tasks that have been executed and their states as well. This execution context must be accessible to every Saga participant, which is why it should be stored in an external system dedicated to execution context management.

In the Saga pattern, the completion of a task triggers the execution of a set of tasks which most times has a cardinality of one. Nonetheless, when the set cardinality is higher than one, the next transactions will be triggered in parallel, leading to parallel execution branches. At some point, even if it is at the end of the Saga, these branches must converge into a single branch. This convergence must be carefully handled. Each branch will finish at different points in time, and the last task of each branch will trigger the execution of the first task of the output branch. However, the establishment of precedence relations should be done in a way that the first task of the output branch waits for all the input branches to complete. Sometimes, it may also be interesting for the execution of the output branch to wait for specific subsets of input branches to complete. Thus, some mechanism to check if a task should be executed is required; the arrival of an event from a previous task is not enough to trigger the execution of a transaction. This mechanism will be implemented by the Asserter module. Further details on this mechanism will be presented in the Section 3.3.1,

Let's dive deeper into the trigger at the completion of a transaction. How does this trigger work? As it was mentioned in Section 2.2.3, tasks will only execute if a triggering event is received. How does this event reach the microservice which executes the transaction, though? A routing mechanism is required if the framework is going to be generic. This routing mechanism would require access to both the execution context and the Saga definition in order to know where the execution is and to determine the next tasks to trigger. Furthermore, this implies that endpoint addresses of the microservices which are responsible for implementing the next tasks must be kept alongside the Saga definition and the execution context. In the Section 3.4.1.1, this routing mechanism is presented under the name of the Router module.

3.2 Representation of Sagas

One of the most fundamental aspects of Shepherd's design is the representation of a Saga, its tasks, and execution contexts. This section is dedicated to their descriptions.

3.2.1 Saga

In this framework, a Saga is defined by a data structure comprised of a graph, and a map referred to as the Saga precedence map. The Saga precedence map is a map where each key is a task name and is associated with a set that contains nested sets of task names.

A graph is composed of a set of nodes and a set of edges. Nodes can be used to represent transactions, and edges to represent precedence relations between transactions. In order to understand where the Saga begins and terminates, a set of nodes that represent start and end-states is required. Differentiating node types can be achieved by labeling nodes. Edges should be directed in order to establish task precedence relations. Also, as compensatable transactions may fail, two

types of task precedence relations must be established, a success and a failure relation. Therefore, graph edges must also be labeled. In order to ease navigability in the graph, edges that indicate what compensatable transactions are compensated by what compensating transactions were introduced to the graph. Lastly, for the Router module, which will be described in Section 3.4.1.1, one more type of node was created to represent the microservices where the tasks to which it is related are implemented. These nodes will be referred to as service nodes. Tasks are connected to service nodes by edges with a label called 'implemented_by'. The 'implemented_by' edges will be used by the Router module to discover which microservice executes each next task and, consequentially, send the trigger events to those microservices.

As it was mentioned in Section 2.2.3, the transactions (or tasks) of a Saga have different types. Four transaction types in Sagas have been discussed: compensatable, compensating, retriable, and pivot transactions. Just like compensatable transactions, pivot transactions can fail. The difference between compensatable and pivot transactions only exists from a business perspective: pivot transactions of a Saga ultimately confirm the business success of a Saga. As this framework's purpose is to implement the Saga Pattern regardless of specific business scenarios, pivot transactions were not considered by the framework. Thus, in the scope of this framework, only three transactions (or task) types are considered: compensatable, compensating, and retriable transactions. In addition, tasks are identified by a name which is also referred to as a task identifier.

Following the previous graph description, a Saga graph contains three types of nodes: task nodes, state nodes, and service nodes. A Saga always starts at the state node 'START', which represents the starting state, and terminates either at the state node 'SUCCESS', in case the Saga succeeded, or the state node 'COMPENSATED', in case the Saga failed and had to be compensated. These three state nodes must be defined in every Saga definition graph. Regarding service nodes, these nodes must contain a property called address which is the real-world address of where the events destinated to these services should be sent. It is a form of service address resolution which allows for the address to be easily changed as needed.

Like it was mentioned at the beginning of this section a Saga is not just defined by a graph. Note that a Saga is also defined by the Saga precedence map. The precedence map will be utilized to specify the required tasks to be completed in order to start the execution of each task. Its use and purpose will be explained in more detail in Section 3.3.1.

Figure 3.1 presents a graph of a Saga. Observe that it is represented in the context of the framework; there are no pivot transactions, and the edges 'compensated_by' and 'implemented_by' are represented.

When a Saga is executed, a different data structure must keep track of the execution of the Saga. This data structure will be used for the framework to decide what to do when executing each task. In this work, this data structure is referred to as Context.

3.2.2 Saga Execution: a Context

A Context is a data structure composed of a graph, a string, three maps referred to as the Context precedence map, the event idempotency map, and the return values map. The Context precedence

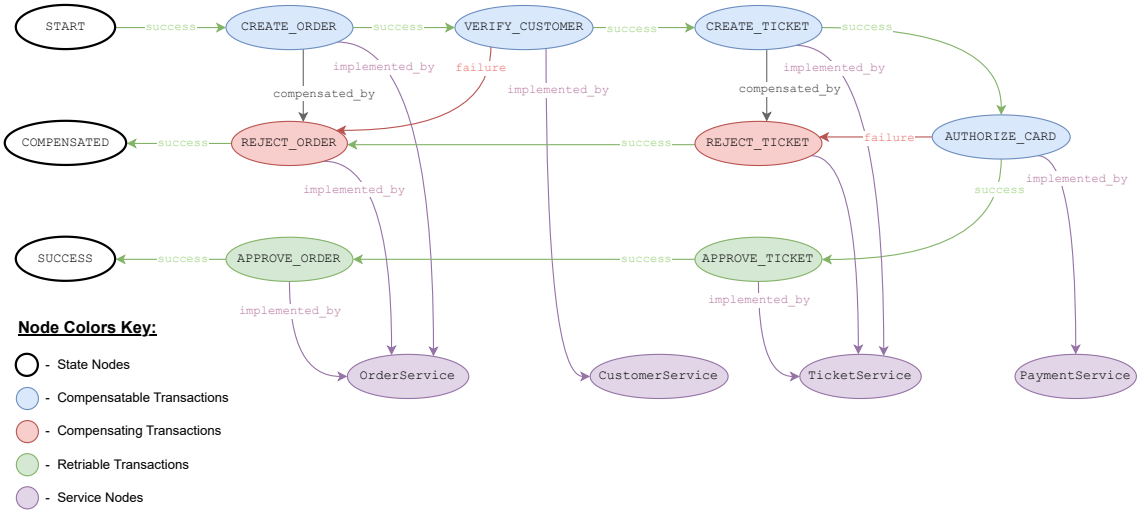


Figure 3.1: A Saga graph.

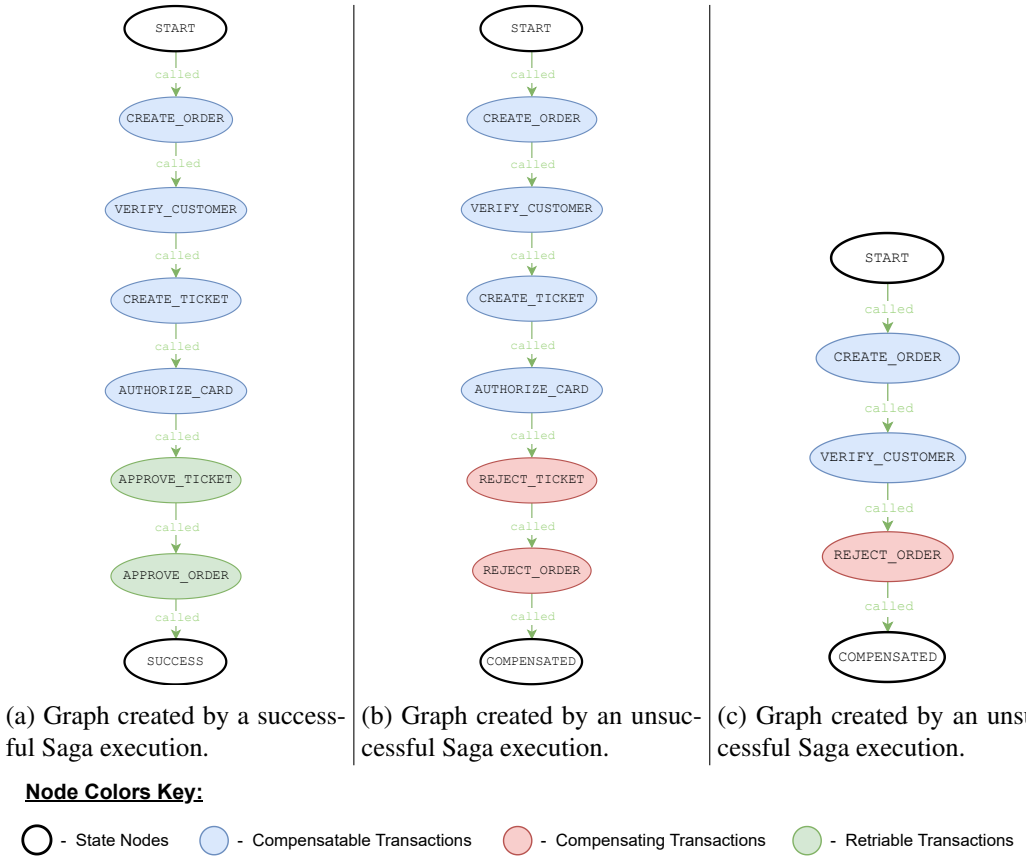
map is a map where each key is a task identifier and is associated with a set that contains task identifiers, whereas the event idempotency map is a map where each key is a task identifier and is associated with a set that contains event identifiers. Lastly, the return values map is a map in which each key represents a task identifier and is associated with two JSON documents. A Context must be kept for every execution of a Saga.

The graph traces the execution of tasks in the Context. It does so by creating nodes for each task as they are triggered. Those nodes are created by consulting the Saga definition graph and querying it for the next tasks given a recently executed task. As for the string that is part of the Context, it indicates the state of the Context. The states in which a Context may be are explained in the Section 3.2.3. The Context precedence map is closely related to the Saga precedence map discussed in the Saga definition. As it was mentioned for the Saga precedence map, these data structures will be discussed in more detail in Section 3.3.1. The return values map is required in order to pass on the results of each task to the next tasks. The map's number of keys must be two times the number of tasks plus two to also store the Saga's input and output. This will be explained in Section 3.3.3. Lastly, the event idempotency map was introduced to the Context data structure as part of a mechanism to guarantee, as the name suggests, event idempotency. This mechanism will be discussed later in Section 3.3.2.

The Context graph is composed of two types of nodes and a single type of edge. The nodes represent tasks or start and end-states, and the edges indicate the order of precedence between nodes, may those be states or tasks. The state nodes are created at the genesis of a Context and are copied from the Saga definition graph. The nodes labels are 'Task' and 'State', whereas the edges label is 'called'. As it will be discussed in further detail in Section 3.2.3, the framework requires a task to have two different state machines. For this reason, besides their identifier, task nodes have three properties: a string specifying the type of transaction and two strings to specify the two states of a task: the technical and the business state. At the completion of a Context, a

path must exist between the state node, identified by the name 'START', and one of the end-state nodes, 'SUCCESS' or 'COMPENSATED'.

The return values map associates every task with two JSON documents, one for its input and another for its output. At the beginning of each task, the framework should retrieve the outputs of all the predecessor tasks and create the input for the current task by merging the previous outputs into a single JSON document. During the execution of the task, the input document can be used as required. When the task execution terminates, the task's output should be stored in the respective JSON document. It can later be consumed either by another task or the framework's interface, which returns it to the client who requested the Saga execution. A further explanation of this mechanism is present in Section 3.3.3.



(a) Graph created by a successful Saga execution.

(b) Graph created by an unsuccessful Saga execution.

(c) Graph created by an unsuccessful Saga execution.

Figure 3.2: Three graphs created by three executions of the Saga presented in Figure 3.1.

Figure 3.2 presents three graphs that could be created by the execution of the Saga defined[2] in Figure 3.1. Bear in mind that these graphs by themselves do not define a Context; the state string, the Context precedence map, the event idempotency map, and the return values map are not shown in the figure. Furthermore, note that the properties of the task nodes are not defined as well.

The graph in Figure 3.2a would be created by the successful execution of the Saga. Observe

[2]Actually, it is partially defined as the Saga is also defined by the Saga precedence map.

that all the compensatable tasks have been executed, and upon their success, the retriable transactions were executed. The other two scenarios refer to different situations. The graph in Figure 3.2b presents the situation where the execution of the task "AUTHORIZE_CARD" does not succeed business-wise, i.e., the card in question was not authorized. Thus, the successful compensatable transactions were compensated by executing their respective compensating transaction in reverse order to the execution of the compensatable transactions, just as it is defined by the Saga definition graph from Figure 3.1. On the other hand, the graph in Figure 3.2c explains the scenario where the transaction 'VERIFY_CUSTOMER' fails (from the business perspective). Here, the task 'REJECT_TICKET' should not be executed as its matching compensatable task 'CREATE_TICKET' was also not executed before. Nonetheless, this conclusion does not have to be deduced on-the-fly by the framework as it is present in the Saga definition graph.

3.2.3 State Machines

Three state machines were defined in the scope of this framework, one for each Context and two for each task. This section is dedicated to their explanation.

Shepherd requires these state machines for two main reasons: control and visibility. Maintaining the states of Contexts and tasks supplies the framework with relevant information for the control of the execution flow. Decisions on where to route events and on what actions to perform are partially based on these state machines. Furthermore, keeping track of the states also enables visibility into Saga executions even while they are running.

3.2.3.1 Context State Machine

As Figure 3.3 shows, a Context may be in 5 different states: 'START', 'IN_PROGRESS', 'SUCCESS', 'COMPENSATING', and 'COMPENSATED'. A Context is in the 'START' state at its creation, before any event is sent, i.e., before the execution of its first task is triggered. While the Context is in execution, it may be in either the state 'IN_PROGRESS' or the state 'COMPENSATING'. The Context enters the 'COMPENSATING' state when a business failure has occurred, which happens when one of its tasks has entered the 'FAILED' business state (which will be discussed in Section 3.2.3.2). Lastly, when a Context terminates, it enters the 'SUCCESS' state or the 'COMPENSATED' state. In order to enter the 'SUCCESS' state, two conditions must be satisfied: a path from the state node 'START' to the state node 'SUCCESS' must exist in the Context graph, and all tasks should have succeeded business-wise, i.e., all tasks should be in the 'SUCCESS' business state. As for the 'COMPENSATED' state, the conditions for a Context to enter it are the following: a path from the state node 'START' to the state node 'COMPENSATED' must exist in the Context graph; one compensatable task should have failed from the business perspective and entered the 'FAILED' business state; the other successful compensatable tasks should have been compensated by their respective compensating transactions. In other words, the remaining compensatable tasks must be in the 'COMPENSATED' business state.

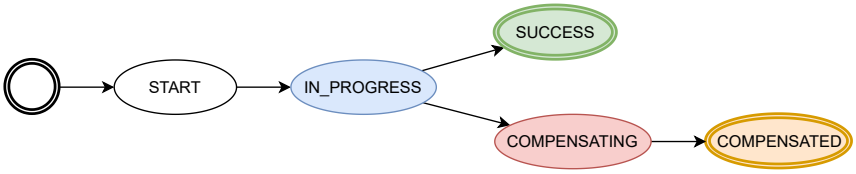


Figure 3.3: State machine of a Context.

3.2.3.2 Task State Machines

As it was mentioned before, tasks are subject to two different state machines. This is required in order to deal with the caveats of the business and technical layer of the Saga pattern, as described in the Section 2.2.3. One state machine defines the technical state of the execution of a task and the other defines the business state. If two state machines are not used, then more complex state machines would result as it would be necessary a combination of states to fully represent the state of a task. For example, as it will be presented in this section, while a task is in a 'ROUTED' technical state, the business state may change. For this reason, two state machines were utilized to determine the state of a task.

Figure 3.4 presents the state machine through which tasks go through when they are handled by Shepherd, from the technical standpoint, independently of the task itself. As the figure presents, there are four states: 'CREATED', 'ASSERT', 'EXECUTED', and 'ROUTED' which is the only end-state. After its genesis, a task is in the 'CREATED' state. As it will be approached in Section 3.4.1.1, tasks are created before routing the events to the microservices that will execute them. When the event is received at the executing microservice, the task will enter the 'ASSERT' technical state, meaning its processing by the Asserter module started. The 'EXECUTED' state is entered after the task is executed and at the beginning of the processing by the Router module. The task enters its final technical state 'ROUTED' when the execution of the Router module is terminated and all the events which will trigger the following tasks have been sent.

In Figure 3.5, the state machine to which a task is subjected, from a business point of view, is presented. It is composed of five states: 'READY', 'IN_PROGRESS', 'SUCCESS', 'COMPENSATED', and 'FAILED'. 'READY' is the starting state of a task, i.e., when it is created it is in this business state. A task transitions to the 'IN_PROGRESS' business state when its execution begins. Business-wise, only compensatable tasks may succeed or fail; retriable and compensating transactions always succeed business-wise. Upon their execution, compensatable tasks may transition to the states 'SUCCESS' or 'FAILED', whereas other tasks always transition to the 'SUCCESS' state. If a compensatable task fails, the Context to which the task belongs enters the 'COMPENSATING' state. When a Context enters the 'COMPENSATING' state, compensatable tasks which are in the 'SUCCESS' business state will eventually get compensated by the execution of their

Figure 3.4: State machine of the technical state of a task execution.

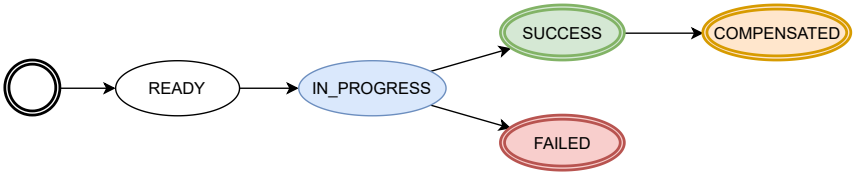


Figure 3.5: State machine of the business state of a task execution.

respective compensating tasks. When that happens, the compensatable tasks transition from the 'SUCCESS' state to the 'COMPENSATED' state.

3.3 Functional Mechanisms

This section is dedicated to describing mechanisms that were developed to offer some of the most relevant features of this framework to its user.

3.3.1 Task Execution Assertion

As mentioned in Section 3.1, a robust definition of precedence relations between tasks is required in this framework in order to support the execution of Sagas with parallel tasks. This need surges when closing parallel branches and the execution of an output branch starts. Observe Figure 3.6. Defining precedence relations just with the graph 'success' relation is an ambiguous definition for the start of the execution of 'TASK_D'. Should the execution of 'TASK_D' start when the event of 'TASK_A' arrives? Or when the events from all the previous tasks arrive? Or when the events from a subset of the previous tasks arrive? Or should more than one situation trigger the execution of the task?

As a solution to this problem, the Saga precedence map was introduced to the Saga definition. Besides defining the graph precedence relations between tasks, it also becomes necessary to define the tasks that must have sent triggering events for a task to begin its execution. This is achieved by defining the sets of identifiers of tasks that must dispatch the trigger events for each task to begin their execution. This definition is accomplished in the Saga precedence map, by associating the said sets with the identifier of the task which gets triggered by one of those sets. It may be necessary to define several conditions to start the execution of a task, i.e., to define several sets of tasks for each task. See Listing 3.1 for an example of this map definition.

With respect to the Context, in order to keep track of the tasks that have sent the trigger events to other tasks, the Context precedence map was created. Upon the arrival of events at the Asserter module, the identifier of the task which triggered the receiver task is registered to the set associated with the receiver task in the Context precedence map. This registration process will be referred to as caller task registration. Then, the framework will calculate the differences between every set associated with the current task identifier in the Saga precedence map and the set of task identifiers associated with the current task identifier. If any of these differences result in an empty set, then

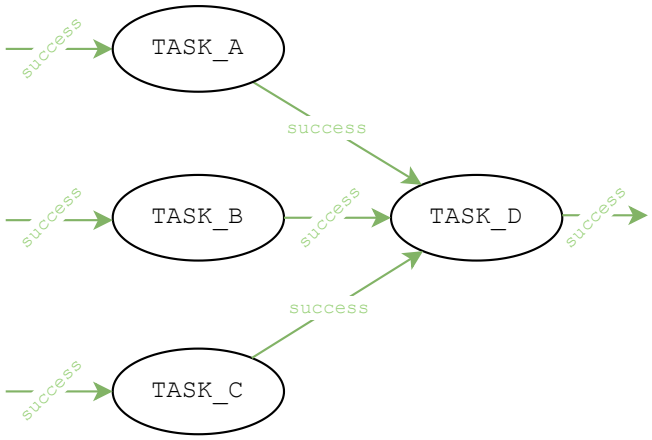


Figure 3.6: Example of the closing branches problem.

the task should begin its execution. The pseudo code in Listing 3.1 expresses this logic in a simple piece of code.

This mechanism also works as a condition for transitioning into end-states. Two entries must be defined in the Saga precedence map to determine the sets of tasks that require to be executed to transition into the end-states 'SUCCESS' and 'COMPENSATED'. The mechanism also creates entries in the Context precedence map for the Context's end-state. Tasks that have Saga graph precedence relations with one of the end-states are registered in the said entries. The form of checking whether the Context should transition into the end-state is partially[3] done using the same set difference operation described before.

[3]The term 'partially' was used because the transition also depends on other factors which are not relevant to this section.

```
Saga_precedence_map = {
    'task_d' = Set(
        Set('task_a'),
        Set('task_b', 'task_c')
    )
}

context_precendence_map = {
    'task_d' = Set('task_b')
}

def should_task_execute(task_id, Saga_precedence_map, context_precendence_map):
    tasks_arrived_set = context_precendence_map[task_id]

    for tasks_set in Saga_precendece_map[task_id]:
        diff = tasks_set - tasks_arrived_set
        if len(diff) == 0:
            return true
    return false
```

Listing 3.1: Task execution assertion logic

For example, in the case of Figure 3.6, let's assume 'TASK_D' wants to know if it should execute or not. Let's consider it should execute when the event from 'TASK_A' has arrived or the events from 'TASK_B' and 'TASK_C' have arrived. Using this mechanism, two sets would be defined for 'TASK_D': a set containing only 'TASK_A' and a set containing 'TASK_B' and 'TASK_C'. Furthermore, let's take into consideration that only an event from 'TASK_B' has arrived. These conditions are represented in Listing 3.1. This mechanism would then calculate the following set differences: Set('TASK_A') - Set('TASK_B'), and Set('TASK_B', 'TASK_C') - Set('TASK_B'). As one can determine, none of these differences would result in an empty set. Thus, in this situation, the mechanism presented in this section would impede the 'TASK_D' from beginning its execution upon the sole arrival of the event from 'TASK_B'.

3.3.2 Event Idempotency Guarantee

In line with what was presented in Section 2.2 regarding the use of event brokers, communication between the microservices implemented with this framework should be done using an event broker. Further details on the event broker's specific requirements are presented in Section 3.4.2.3. The event broker used for the implementation of this framework must guarantee an *at-least-once* message delivery taxonomy, or ideally, an *exactly-once* taxonomy. However, since very few event brokers guarantee the *exactly-once* taxonomy, let's stick with the *at-least-once*. This implies that events might be received several times by a task. This is clearly a problem if not handled correctly; a task may execute more than once and leave the system in an inconsistent business state. Thus, a mechanism to guarantee the event idempotency is required by the Shepherd framework.

The idea behind it is simple: when dispatching an event, the sender task lets the receiver task know that the event was sent and that it should expect its arrival. When the event arrives at the receiver task, it knows that the event is expected and marks it as received. Repeated events will not be processed because only the first event was expected.

This mechanism introduces the event idempotency map to the Context data structure. This map, where each key is a task identifier and is associated with a set that contains event identifiers, is used by the receiver and sender tasks so that they can jointly keep track of the expected events. The sender task adds the identifier of the event it is going to route to the set associated with the task identifier that will receive the event. Upon the reception of the event, the receiver task checks that the event identifier is present in the set associated with its identifier and deletes it from the set. If the receiver task is not able to validate that the event identifier is present, then the task execution will not proceed.

3.3.3 Task & Saga Return Values

It is a business requirement of this framework that Sagas must be able to receive inputs and produce outputs. For that, a mechanism needs to be in place to transfer values between tasks, from the framework's client[4] to the first task of a Saga, and from the last task to the client.

The mechanism works by constructing the input of a task before executing it and by posting the output upon its completion. For each task, two JSON documents are required, one for the task's input and another for the task's output. The construction of the input document of a task functions as follows: before executing a task, the output documents from the preceding tasks are merged into one JSON document in which each key is the name of a preceding task and is associated with the respective output document. Having constructed the input document, the task can utilize the document as required in its execution. The task should post the object or value it is supposed to return to its output document. Having posted the output, the following tasks can collect the task's output from its output document and construct their input documents. Bear in mind that the input and output documents are stored in the Context return value map so that they are accessible by every task of the Context.

When delivering input and output documents from and to the framework's client, some aspects must be taken into consideration. The Interface, which receives the request from the client to execute a Saga, before triggering the first tasks of the Saga should post the client's input document to the 'START' state node's output document. Having posted the client's input to that output document, the first tasks will be able to construct their inputs based on the starting state node's output document. Regarding the dispatch of the Saga's output back to the framework's client, after a Context is terminated as described by the process in Section 3.4.1.1, an event is sent to the Interface informing the Saga has been terminated. Upon the reception of this event, the Interface can collect the output documents of the tasks that have a precedence relation to the end-state in the Context graph and then send a merged JSON document (constructed using the same method

[4]The framework's client is the entity which performs Saga execution requests to the Interface.

as with the construction of input documents, to the framework's client). In the request to execute a Saga, the client should specify an endpoint's address to receive the output from the Saga if there is any. The framework will store this endpoint's address so that the Saga's output document can be delivered via an HTTP request.

3.4 Architecture

In this section the architecture of Shepherd is discussed in more detail, as well as the different components of the framework. Figure 3.7 presents a high-level design of a system's architecture using the framework executing a Saga with five tasks, where three tasks are executed in parallel. It can also be observed that the system is composed of four microservices and that each microservice executes one task, with the exception of microservice '2' which executes tasks 'B' and 'C'. Note that each task is built by four components: the Expecter, the Asserter, the Task Core, and the Router. Each component will be explained in more detail later in this section. Bear in mind that the Task Core is the only component not to be implemented by this framework; its implementation is specific to the application of the framework, meaning the Task Core should be implemented by the framework's user. Also, note that there are two external systems to the microservices: the Context Manager, and the Interface. Furthermore, the queues present at the entrance of each microservice are implemented by an external event broker and were represented in this way for the sake of simplicity of the representation.

As it can be observed in Figure 3.7, the framework's logic execution is distributed across the microservices, encapsulating the execution of tasks. The alternative considered to this approach was to have a central orchestrator that would process all the framework's logic. However, the proposed architecture allows for the computational load to be distributed among the framework's participants, eliminating the need to build a central orchestrator. Not having a central orchestrator was estimated to reduce operational costs of the framework as one less component requires hosting.

The main communication entity which is sent from and to the microservices operating under this framework is an event with the structure presented in the Listing 3.2. It is composed of a unique identifier, an object that defines the task that should receive the event, and another object that defines the Context it was triggered in. The object which defines the said task contains four properties: an address to know where the event should be sent; a type that defines the transaction type as discussed in Section 2.2.3; the name of the task whose execution created the current event; and lastly, the name of the task the event triggers. There is a specific situation where only the name of the task is used from this object[5]. On another note, the object which defines the Context possesses only two properties: a unique identifier of itself, and the name of the Saga definition it is executing. Both the unique identifiers of the event and the Context object should be randomly generated. In the Listing 3.2, identifiers are composed of sixteen random alphanumerical characters.

[5]This situation is specified in Section 3.4.1.1.

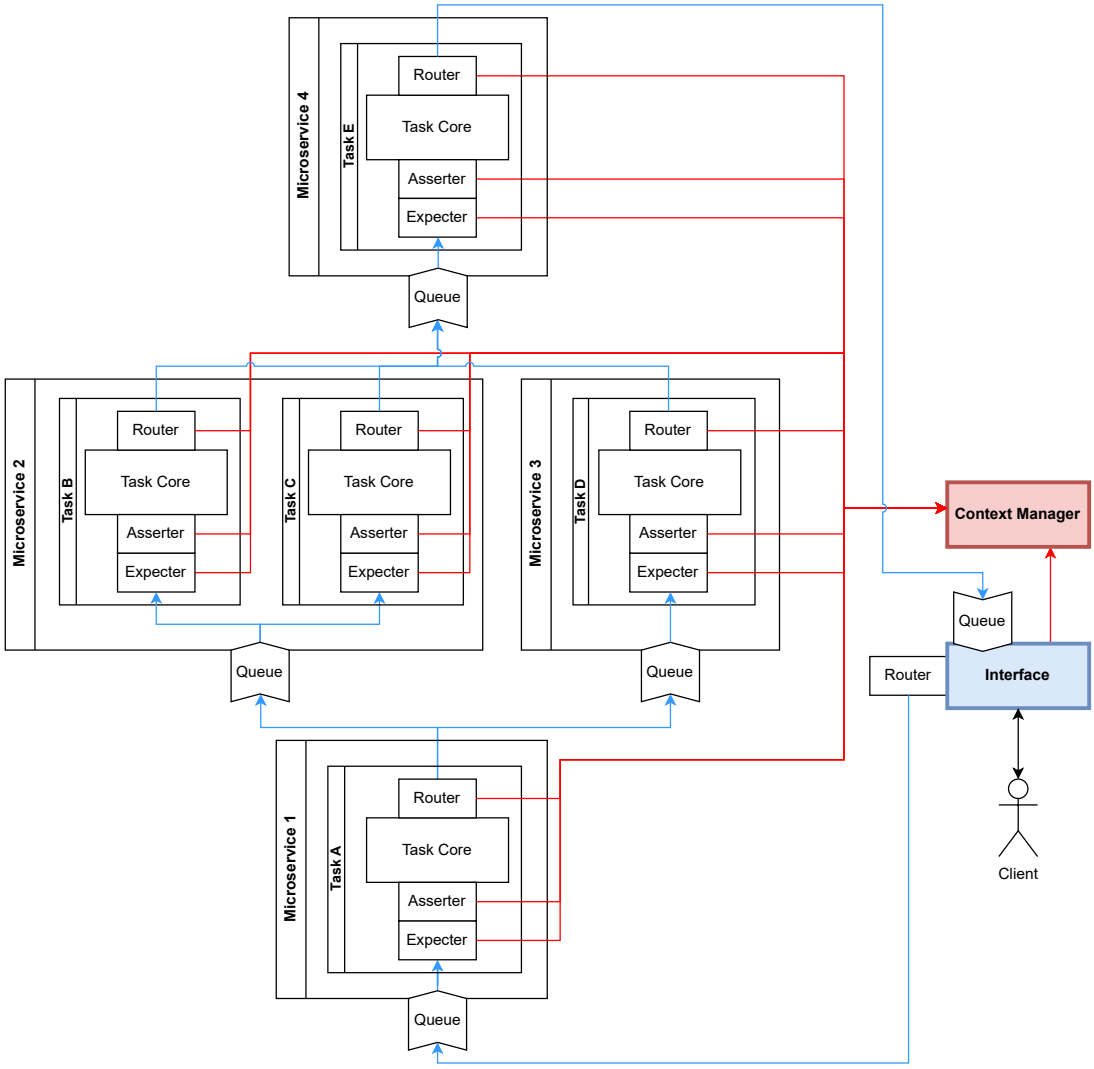


Figure 3.7: High-level architecture design of a system implemented by the framework.

The event is exchanged between the microservices (and also the Interface). The blue lines from Figure 3.7 represent the channels through which it flows. The components described in Section 3.4.1 all take as input and output the event, with the exception of the Task Core.

```
{
    "id": "1WAmN6XWNoidn4u7",
    "receiver": {
        "address": "a-queue-identifier",
        "type": "COMPENSATABLE",
        "callerName": "A_FIRST_TASK",
        "name": "A_SECOND_TASK"
    },
    "context": {
        "sagaName": "a_saga",
        "id": "qSUPXiOoHo7VPg49"
    }
}
```

Listing 3.2: Event Structure

3.4.1 Task Components

Besides the Task Core that implements the business logic, there are three software components that are used in the execution of each task as Figure 3.7 suggests. These units of software are replicated throughout every microservice and encapsulate the Task Core execution. The Task Core is specific to every task, as it is where the task business logic and implementation reside. In each task execution, the components are invoked in the following order: Expecter, Asserter, Task Core, and lastly, the Router component. Note that the Router is also replicated in the Interface and that the first component invoked in a Saga execution is the Router replica from the Interface.

Take into consideration that all components are executed in the Context of a task (except for the Interface's Router replica). When the term 'current task' is used in the following sub-sections, it is meant the task to which an instantiation of the component being discussed belongs. Furthermore, consider that tasks are always executed inside a Context. Thus, whenever a Context is referred to, it is the current task's Context that is being referenced.

3.4.1.1 Router

The Router component is responsible for understanding what the next tasks in the Context are and for creating and routing events to the microservices that will execute those tasks. Besides the event, the Router component receives as input a boolean which indicates the business success of the Task Core executed before its invocation and produces a list of events to be routed as output. The execution of the Router is divided into three stages. The first stage is dedicated to Context and task state updates; the second stage is about creating new tasks and events; lastly, the third stage executes only if the Saga is to terminate and is dedicated to the formal termination of the execution of a Saga.

As it was mentioned, the first stage is dedicated to state updates. The Router component begins this stage by setting the technical state of the task that was triggered by the event received as input to 'EXECUTED'. Afterward, the business state of the recently executed task should be set accordingly to the boolean received as input that indicates the business success of the task. If the task succeeded business-wise, then it should check whether the task that was executed compensated any compensatable task and if so, the business state of the compensated tasks should be set to 'COMPENSATED'. On the other hand, if the task has failed, the state of the Context should transition to 'COMPENSATING' and all the compensatable tasks of the Context that do not have a matching compensating transaction should have their business states transition to 'COMPENSATED'. This last step is required for the termination of the Sagas executions.

For the Interface's Router replica, the first stage is somewhat different. Since it is running before any task has been executed, the object which defines the current task in the event (shown by Listing 3.2) only contains the name of the current task, which is set to the name of the initial Saga state, called 'START'. In this situation, the Router component will only set the state of the Context to 'IN_PROGRESS' and not do anything else.

The second stage of the Router component deals with the creation of the next tasks and events of the Context. It starts by consulting the Saga definition to determine what the next tasks in the Context are. This is determined based on which task was executed right before the Router was invoked and on that task's business success. While retrieving the next tasks, the Router also retrieves the service nodes associated with those tasks which contain the real-world addresses to where the events that trigger those tasks should be sent. Having determined what the next tasks are, the component creates them in the Context graph. Afterward, the Router creates the events that will trigger the execution of the new tasks. Note that all the information required by the event data structure presented by the Listing 3.2 was gathered by the Router. Before proceeding to the next stage, the Router registers that the created events should be expected by the respective tasks. This registration is done in the set of event identifiers associated with the current task identifier in the Context's event idempotency map[6]. The created events are then outputted from the Router component so that they can be sent to their respective destination addresses. The events dispatch is not handled by the Router itself, it just creates the events to be sent. However, after the dispatch, the current task should transition into the 'ROUTED' technical state, and that transition is the Router's responsibility.

Some aspects of this stage's execution differ when it is run by the Interface's Router replica. There is no success boolean as input to the stage in this situation as no Task Core was executed beforehand. Thus, when consulting the Saga definition for the next and creating the tasks in the Context graph, the state node identified by the name 'START' is used. In other words, the Router checks what are the tasks associated with the 'START' state node in the Saga definition graph and creates them in the Context graph, associating them with the same state node. The same process is then carried out with these newly-created tasks: events are created with them and the registration of the new events is done as well.

[6]Note that this registration is part of the event idempotency guarantee mechanism described in Section 3.3.2.

The third stage formally terminates the execution of a Saga and is only executed when no tasks and events were created in the second stage. It starts by checking in which state the Saga can terminate. As it was discussed in the Section 3.2.3.1, the only two valid end-states of a Context are 'SUCCESS' and 'COMPENSATED'. If the Context is in the 'IN_PROGRESS' state, then it must end in the 'SUCCESS' state, whereas when the Context is in the 'COMPENSATING' state, it must terminate in the 'COMPENSATED' state. At this point, it is assumed that the last task of the Context was already executed and that it is not possible to transition from the 'IN_PROGRESS' state to the 'COMPENSATING' state. This assumption is not problematic because a later verification will check if more tasks will execute or not. Afterward, the current task gets registered in the Context precedence map entry for the determined end-state as described in Section 3.3.1. Upon the task registration, the Router validates whether the transition to the end-state should occur and the validation is different for the two possible end-states. What is common to both transitions is that the assertion mechanism presented in Section 3.3.1 must verify that the transition to the end-state is valid. Besides that, in order to transition to the 'SUCCESS' end-state, all the tasks of the Context must be in the 'SUCCESS' business state and 'ROUTED' or 'EXECUTED' technical state. Regarding the transition to the 'COMPENSATED' end-state, besides the assertion mechanism validation, the Context may perform the transition if there are no compensatable tasks in a business state different from 'COMPENSATED' and all tasks are in a 'ROUTED' or 'EXECUTED' technical state. In case any of the transitions are allowed, the Context is formally terminated by creating a 'called' graph relation between the current task and the end-state, and by transitioning the Context state to the end-state. Lastly, as described in Section 3.3.3 an event is sent to the Interface informing it that the Saga execution terminated for it to send the Saga's return value to the client.

3.4.1.2 Expecter

The Expecter component's purpose is to implement the event idempotency guarantee mechanism presented in Section 3.3.2. It is responsible for guaranteeing that one event only triggers one task execution. As Figure 3.7 indicates, the Expecter component is the first component to receive the event and, for this reason, dictates if the event is forwarded to the other task components or not.

The actions the Expecter component performs are described in Section 3.3.2: upon the reception of an event, it will consult the set of event identifiers associated with the identifier of the current task in the Context's event idempotency map and verify whether the identifier of the received event is present in the set. If it is not, the event is not expected, and the rest of the task is not executed. However, if the event is found in the set, it will be deleted so that that event is not considered as expected anymore. If the same event is received again by the Expecter component, it will not be expected, and thus the task will not be executed.

3.4.1.3 Asserter

The main purpose of the Asserter component is to implement the task execution assertion mechanism, described in Section 3.3.1. The execution of this component is divided into two stages: firstly, it verifies if the conditions for the execution of a task are met and, afterward, it constructs the input document to the task. Regarding the execution conditions, besides performing the verifications of the assertion mechanism, it also checks whether the execution of the task should be canceled.

Two verifications are done by this module in order to determine if a task should execute. The first stage of the asserter begins by transitioning the task from the 'CREATED' technical state to the 'ASSERT' technical state. Afterward, as described by the assertion mechanism description, the caller task registration is done in the Context assertion map (see Section 3.3.1). Upon the completion of the registration process, the set which contains nested sets associated with the identifier of the current task in the Saga precedence map is be iterated through in order to check if all tasks of any of those sets have been registered by the caller task registration process. This verification is done with a set difference operation, as described by the mechanism section. If no set is found, the Asserter module will not allow the task execution to proceed, i.e., the task execution must wait for the arrival of the events of the required tasks that fulfill one of the execution conditions defined in the Saga precedence map. Furthermore, a second verification is done by the asserter module. It verifies if the task execution should be canceled or not. If the Context is in the 'COMPENSATING' state, only 'COMPENSATING' transactions should be executed. In this situation, the execution of transactions of any other type should be canceled. In any other situation, tasks are not canceled.

The second stage of the Asserter component constructs the input document to the task. As mentioned in the description of the return values mechanism (section 3.3.3), before the execution of a task begins, the input document is constructed. That is done by gathering the output JSON documents from the previous tasks and combining them into one JSON document where each key is a task identifier that produced an output and is associated with the respective output JSON document. The new JSON document is then stored in the Context return values map, and it is available to be consumed as required by the task.

There is a serializability requirement for the execution of the Asserter. Consider the following scenario: the execution of task A can only be triggered upon the reception of two events: one from task B and another from task C. If these two events were to trigger the execution of task A at the same time, the Asserter components should not execute in parallel; their execution should be serialized. If the two concurrent Asserter components were executing, there would be the possibility of both performing the caller task registration at the same time. When each of them verifies whether all required events have been received to proceed with the execution, both of the Asserter components would confirm that the task can execute. Thus, the tasks after task A would be triggered twice, which, obviously, is not correct. By serializing the execution of the Asserter component in each microservice, this problem is solved.

3.4.1.4 Task Core

The Task Core is the component where the task implementation resides. The Task Core is programmable and is different for every task definition. Its purpose is to allow the framework's user to implement the business logic the task is supposed to perform, and with that to implement Sagas' business logic.

Because the scope of this work is to implement this framework using MuleSoft, as it will be discussed in Section 3.4.3, the implementation of a Task Core is done by creating a Mule Flow for the task. The framework will execute the Mule Flow when allowed by the Expecter and Asserter components. This means that the task can be implemented using any of MuleSoft's capabilities. Furthermore, bear in mind that the input document of the task can be consumed as needed, and the values that the task outputs should be stored in the task's JSON output document.

3.4.2 External Systems

This section is dedicated to the description of the three external systems identified at the beginning of Section 3.4.

3.4.2.1 Context Manager

The Context Manager is a database system whose purpose is to store Saga and Context data structures and to make them available for microservices to access. It provides a shared storage system for microservices; in a way, it can be seen as a communication means between them. The Context Manager is what keeps track of the state and history of executions. It is what the framework holds as truth and the basis for all its functioning. For this reason, the Context Manager is also a single point of failure of the system.

Given its nature, the requirements for this system are very demanding. The main requirements for the system are five-fold:

1. **Highly Available** - Since the Context Manager is a single point of failure of the framework, the risks of it being unavailable should be mitigated as much as possible. To do so, the Context Manager should be implemented by a technology that allows a form of clustering or distributed storage, which increases the robustness of the system to failures.

2. **Performant** - This storage system is invoked several times per task execution. Each call to the system introduces an overhead to the task execution which should be minimized.

3. **Scalable** - The Context Manager should be able to scale as required by the system which utilizes the framework. The bottleneck of the framework's scalability is the Context Manager's scalability.

4. **Persistent** - In case of total failure of the Context Manager, it should be possible to recover its contents and minimize the loss of data.

5. **Graph Compatible** - Since Contexts and Sagas are partially defined by graphs and their manipulation and consultation are done very often, the Context Manager should offer graph storage and manipulation capabilities. Without these, a significant overhead would be introduced by serializing and deserializing the Sagas and Contexts graphs each time they would be accessed.

3.4.2.2 Interface

The Interface is an HTTP REST API which is the point of interaction with Shepherd. Its purpose is to enable other systems and services to execute Sagas for them with the layer of abstraction that APIs provide. By performing a Saga execution request to the API, the client does not need to be aware of the complexities involved in the implementation of the Saga Pattern; the client is just interested in the execution of a Saga as a whole to deliver business value for them.

The Interface's REST API should have 2 endpoints: one for executing Sagas and another for querying the status of a Saga execution (what this framework refers to as a Context). The Saga execution request is composed of the name of the Saga to execute, the input to the first task of the Saga, and an HTTP address to which the Interface can send the output of the Saga. Upon the completion of the Saga execution, the API will send the Saga's output via an HTTP request to the HTTP address the client specified in the Saga execution request. Regarding the status query request, the client provides the name of the Saga and the identifier of the Saga execution (the Context's identifier) to get information on its status. A standard HTTP response with the Context state string is sent to the client.

3.4.2.3 Event Broker

Event brokers are utilized in order to reliably transmit events according to a pre-established set of guarantees. Their purpose is for distributed systems to abstract themselves from the complexities of communication. Shepherd uses an event broker as the primary means of communication between microservices.

The framework has specific requirements for the event broker. As Figure 3.7 suggests, each microservice (and the Interface) must have a queue to which they are the unique subscribers. Moreover, as mentioned in Section 3.3.2, the event broker should offer an *at-least-once* event delivery guarantee. Some event brokers offer the *at-most-once* delivery guarantee, which would not work with this framework as events could be lost and, as a result, Sagas executions would be halted permanently. Besides that, the event broker should also guarantee that the order of the events is maintained at their reception by the microservices. Lastly, the selected event broker should be compatible with MuleSoft so that queues can be subscribed by a Mule flow. The flow can then execute the components of a task and the task itself.

Another interesting feature for the event broker to have would be serving as permanent storage for the events even after their consumption. This feature is very interesting because it opens the

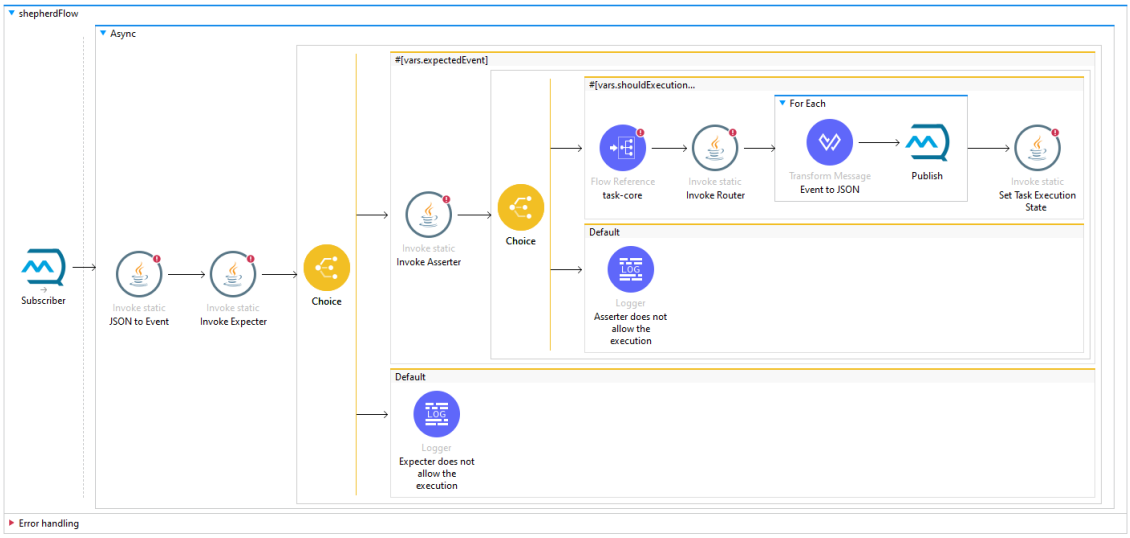


Figure 3.8: Main flow design in Anypoint Studio.

possibility of performing event sourcing as a form of recovery in case of total failure from the Context Manager.

3.4.3 Task Implementation in Mule

In order for a Mule microservice to participate in the framework, it must implement a Mule flow that executes a task upon the reception of an event. For receiving events, the microservice must have a queue dedicated to itself in the event broker. The Mule flow should, as a source, subscribe to the queue dedicated to the microservice, so that it is triggered upon the sending of events addressed to the microservice.

When an event is received, the Mule flow invokes the Expecter component, whose output is a boolean that determines whether the task execution should proceed or not based on the idempotency verifications it is responsible for. If the task execution proceeds, a similar process is done with the Asserter component, it is invoked and its output is a boolean which determines if the task execution should continue based on the Task Execution Assertion mechanism. If, again, the component allows the task execution to continue, the third component, which is the Task Core, is invoked. This invocation has some nuances that will be discussed later. Having executed the Task Core component which outputs the business success of the task in a boolean, the Router component is invoked. Observe that, besides the event, the Router takes as input the business success of the task and produces a list of events to be sent as output. This list must be iterated through and the events must be sent to their respective destination addresses, which are present in each event data structure, shown in Listing 3.2. Figure 3.8 shows the graphical implementation of this flow in MuleSoft's Anypoint Studio.

As a microservice can be responsible for the execution of several tasks, the invocation of the Task Core component is not as simple as it seems. As described in Section 3.4.1.4, the Task Core

component is implemented by the framework's user defining a Mule flow for the task's business logic implementation[7]. As a microservice can implement several tasks, several task flows may exist when invoking the Task Core component. This creates the problem of deciding which task flow should be invoked by the main flow. However, the problem can be solved with a simple flow naming rule. If the task flows are named with the name of the task they implement, then the main flow can just invoke the flow with the same name of the task whose execution the event requests. It is the responsibility of the framework's user to ensure that the name of the task in the Saga definition is the same as the name of the task flow which implements it.

Observe that since the Expecter, the Asserter, and the Router components are all implemented in Java, their invocation is done by Mule's Java connector.

3.5 Shared Responsibility Model

The implementation of Sagas using the Shepherd framework follows a shared responsibility model between Shepherd and its user. This model is the basis for the configuration-first approach to the user experience of the framework. This means that by using Shepherd, users will mainly just configure its execution, and not necessarily build new applications or integrations. However, observe that the end effect of using Shepherd will be the same as if they were building new integrations.

As described in this chapter, the framework is responsible for:

- executing Sagas when requested;
- invoking tasks and deciding when they should run;
- the communication between the microservices;
- managing Contexts;
- dealing with technical errors.

For the successful usage of this framework, Shepherd expects that the user:

- defines a Saga according to what was presented in this chapter;
- defines the Saga's compensation strategy, which is part of defining the Saga;
- defines the Saga's tasks implementations;
- configures the event broker;
- integrates the services that execute the tasks defined in the Sagas into the framework.

By following this shared responsibility model, the framework guarantees that the implementation of the Saga Pattern is successful.

[7] Since two different Mule flows are being referenced, the flow which receives the event will be called 'main flow' and the flow which implements the Task Core will be referred to as 'task flow'.

Chapter 4

Validation

This chapter describes the validation of Shepherd. This chapter is divided into four sections. Firstly, the validation setup is discussed, followed by descriptions of the two validations performed to the framework. The chapter concludes with a discussion of the validation's results.

The two validations were designed to verify whether Shepherd meets its purpose and if it is possible to address the research objectives stated in Section 1.3.

4.1 Validation Setup

In this section, the setup of a proof of concept (PoC) of Shepherd is presented. The implementation of Shepherd built for these validations did not contain all the features presented in the previous chapter; it contained just the necessary features to prove that the framework works. Thus, the implementation of Shepherd will also be referred to as PoC in this chapter. The following section mentions the limitations that were put in place for the PoC of the framework.

4.1.1 Proof of Concept Definition

Some setting restrictions were considered for implementing the PoC of the framework. For this implementation, three limitations on Shepherd's design were set deliberately.

The Return Values Mechanism presented in Section 3.3.3 was not implemented in this PoC. This decision was consciously taken based on two factors. The first one was due to time constraints in the development phase. Moreover, the Return Values Mechanism is a business-level requirement: their use and added value are not in the scope of this work's purpose[1]. As stated before, this work's purpose is to validate whether it is possible to create a MuleSoft framework that implements the Saga Pattern using microservices choreography in a generic form.

The conditions under which this PoC was executed were substantially different from a real-world execution scenario. The PoC's execution was hosted by the author's personal computer, meaning all the framework's components were hosted there as well. This restriction was set due

[1]This work's purpose is to provide an implementation of the Saga Pattern in MuleSoft, regardless of the business context. In order to execute Sagas and perform the validations in this chapter, the business logic of tasks is not required

to time and resource constraints. Furthermore, it was assumed that technical failures would not occur while executing Shepherd. It was supposed that no other actors communicated or interfered with the framework's components besides the two requests that the framework's Interface allows. This led to an ease of the Context Manager requirements. For this proof of concept, the Context Manager did not require to be highly available and fault tolerant. Also, as the deployment environment is so different from a real-world deployment scenario, the scalability requirement was also put aside.

In order to meet the serializability requirement of the asserter described in Section 3.4.1.3, a limitation to the parallelization of task executions *internally* to each microservice was put in place. Due to time constraints, instead of serializing just the execution of the Asserter component, it was decided to serialize the whole task execution. This implies that each microservice can only execute one task at a time. Besides meeting the Asserter requirement, this limitation solely affects the performance of the framework.

4.1.2 Technology stack

This section is dedicated to describing technological decisions for each component of the architecture presented in Section 3.4. Since the purpose of this work is to develop a framework for MuleSoft, some technological decisions are already taken by default, specifically regarding the four task components.

4.1.2.1 Task and its Components

As discussed in Section 2.3, Mule Runtime is a Java-based runtime environment where Mule applications run. Mule applications implement microservices in MuleSoft. Since tasks are executed by the microservices, these must be implemented in Mule. Mule provides a platform for implementing business logic using a vast range of external system connectors and also the possibility of invoking programming languages, such as Java, JavaScript, Python, and others. Thus, in spite of the task and the microservice being implemented in Mule, the task components were implemented in Java with the exception of the Task Core.

As described in Section 3.4.3, a task execution by Shepherd is defined by a flow where the four task components (the Expecter, the Asserter, the Task Core, and the Router) are invoked. In order to allow the framework to be seamlessly used by Mule developers, it was decided that the Task Core was to be implemented in Mule by the framework's user. Since the Task Core is where the business logic of the task is defined, developers keep the responsibility of developing the business logic of the task and, therefore, of the Sagas. The remaining components (the Expecter, the Asserter, and the Router) were implemented in Java due to the complexity of their design. Their implementation in Mule would result in very complex Mule flows which would be very hard to build and read, whereas in Java, complex logic is more readable and understandable.

4.1.2.2 Context Manager

In Section 4.1.1, the requirements for the PoC's Context Manager were discussed. Because of limitations to the environment in which Shepherd was tested, the design's Context Manager requirements from Section 3.4.2.1 were eased.

Redis was the technology of choice for the implementation of the Context Manager. It is an open-source in-memory key-value store database and is compatible with graphs. Because of its support for a cluster deployment topology, Redis allows for horizontal scaling and offers some guarantees regarding availability. Redis is a very performant technology because of its in-memory nature: it stores its contents directly in RAM memory which results in a performance that is orders of magnitude superior to traditional disk databases. Another reason for the choice of Redis is that it supports several datatypes out-of-the-box, which is very useful for storing the Saga and Context data structures that were discussed in the previous chapter. Besides that, a very simple setup was required for its usage in the PoC of the framework. The standard Redis deployment is not a cluster topology, but rather a standalone instance serving as a database over a network. Furthermore, a Java client library called Jedis was used for establishing connections to the Context Manager from the task components.

Redis supports graphs via its module called 'RedisGraph'. RedisGraph enables the storage and manipulation of graphs with high performance. The language used to interact with the graphs is the Cypher graph query language. All graph-related actions are done by using it: querying a graph for specific nodes or relations, creating nodes and relations, setting node or relation properties, and others.

4.1.2.3 Interface

Being an HTTP REST API, the Interface was implemented in Mule. MuleSoft's purpose is to expose assets and services of enterprises via APIs and therefore, it was a straightforward choice as the technology to implement the Interface. Both endpoints defined in Section 3.4.2.2 were implemented. Observe that, since the Return Values Mechanism was not implemented in this proof of concept, the Saga execution request is solely composed of the Saga identifier that should execute.

4.1.2.4 Event Broker

As discussed in Section 3.4.2.3, the choice of the event broker must address the following requirements: *at-least-once* delivery guarantee, compatibility with MuleSoft, maintenance of the order of events, and the creation of an event queue per microservice. MuleSoft's Anypoint Platform offers an event broker which fulfills the four stated requirements: Anypoint MQ. Anypoint MQ is a message broker that can be used to exchange JSON documents using queues. It offers the *at-least-once* delivery guarantee, and since it is developed by MuleSoft, it is fully compatible with Mule and its setup is simple. Also, it has the capability of maintaining the order of events in which they were sent.

4.1.3 Framework Usage

A template for the creation of Mule microservices that participate in the framework was developed. The template contains all the necessary logic in order for a microservice to be a Saga participant. The flow that is responsible for executing the four different task components, described in Section 3.4.3, is present in the template. The template imports the necessary Java code via a Maven[2] package: a process that will be described later. Furthermore, the template contains a set of properties where the connection credentials to the event broker and the name of the queue dedicated to the microservice must be configured. The connection to the event broker is already set up, the developer just needs to configure access to the event broker. As for the implementation of the Task Core component, an example flow that implements it is provided in the template. As stated before, the developers must take into consideration that the name of the flow responsible for the implementation of a certain task should be the same as the name of that task.

As it was discussed, the task components which are not implemented by the developer (the Expecter, the Asserter, and the Router) are implemented in Java. While the PoC was in development, there was a need to update the Java code of those components in several microservices each time a change was made to them. In order not to rely on performing manual changes throughout the several instances of Java code in the different microservices, the Java code which implemented the task components was packaged as a Maven dependency and imported by the microservices. This way, at the point of building the microservices, the Java code was loaded from the Maven dependency. Just like that, changes in Java code would just have to be updated once, and the microservices would update their code by performing the build process.

Anypoint Exchange has the capability of publishing Mule Projects templates, as described in Section 2.3. Also, it has the capability of publishing assets within a specific business group for only the members of an organization to access it. The discussed template was published to a closed business group of Deloitte.

The main principle for the user experience development of the framework was minimizing programming and maximizing configuration. This is what the template tries to achieve: in order to create the microservices, the user only needs to program the business logic of the tasks that compose Sagas. The user also requires to create the definition of the Saga that should execute. However, that is not programming, it is a configuration of how Shepherd will manage its activities.

As it was explained in Section 3.2.1, a Saga is comprised of a graph and the Saga precedence map, whereas a Context is defined by a graph, the Context precedence map, and the event idempotency map[3]. The graphs are stored using the RedisGraph module. Their creation and manipulation are done using the Cypher graph query language. Since Redis is a key-value store database, all of its contents are, in a way, defined as a map. Thus, the Saga and Context precedence maps and the event idempotency maps entries can be stored directly in Redis by carefully managing the key names. Listings A.1 and A.3 are examples of Sagas definitions using Redis commands. In this

[2]Maven is a dependency manager used for importing libraries and external resources to, most commonly, Java projects, but also Mule projects.

[3]The return values map is not used in this PoC as it was described in its definition section.

PoC, the Saga definition from Section 3.2.1 was utilized as it is. Therefore, the framework's user should ensure that the Sagas are defined according to what is presented in that section.

4.2 Validation 1: Framework's Behavior

This validation's purpose is to verify whether the framework behaves as expected. In other words, this validation checks if the PoC implements the Saga Pattern or not. This is done by confirming the Saga execution side effects, such as Context and tasks creation.

4.2.1 Method

This validation lies in the assumption that if a determined set of verifications on the side effects of a Saga execution is successful, it can be assumed that the execution of Sagas was implemented successfully. By performing this validation with several scenarios of different complexities, it is possible to state with a higher degree of confidence that the execution of Sagas was implemented successfully.

After the execution of each scenario presented in Section 4.2.2, the following aspects were validated to confirm whether the behavior of the framework was as expected:

1. **Context Creation** - Validate whether the Context data structure was created in the Saga execution.
2. **Tasks Creation** - Check if the required task nodes have been created in the Context graph.
3. **Context Final State** - Confirm whether the Context is in the expected end-state.
4. **Task Final States** - The technical and business states of every task of the created Context should be at their expected end-states.
5. **Caller Tasks Registrations** - Confirm whether the expected preceding tasks are present at each caller tasks set in the Context precedence map. Note that the expected preceding tasks are defined by the Saga precedence map.

The Redis commands required to check each aspect are present in appendix B.

4.2.2 Scenario Descriptions

For this validation, a total of ten validation scenarios were used to validate the framework. Six of these were created based on two business cases. The remaining four were created by a random Sagas generator in order to ensure that the framework's design is robust. Each of these scenarios was executed by the framework and was validated whether the expected execution side-effects can be observed after their executions. The exact validations that were performed were explained in the previous section.

Like it was mentioned before, in this PoC, Sagas were run on the author's personal computer. For all Sagas, one Mule app and one Redis server were utilized. As discussed, the Context Manager and the Interface were implemented by the Redis instance and the Mule app, respectively. For each Saga, assuming 'n' is the number of services required for executing a Saga, 'n' additional Mule apps were created using the template presented in Section 4.1.

4.2.2.1 Order Food Saga

Consider a food ordering app built with microservices. Let's assume the app has scaled successfully and because of its immense size, performing a food order requires executing transactions across four microservices: the order service, the customer service, the kitchen service, and the payment service. The services' names are suggestive of the business domains they are responsible for. However, a description of each follows: the order service is responsible for managing food orders and tracking them; the customer service stores customer's details and can check whether consumer information is correct; the kitchen service is the point of contact of the restaurants that produce the food that gets ordered and creates time estimates for the conclusion of the food preparation; the payment service is responsible for billing the customers. Note that this business scenario was taken from [12].

The Saga constructed for this business scenario performs a fictitious food order in the described (also) fictitious application. The graph of this Saga's definition is present in Figure 3.1. As this Saga was taken from [12], the following is a transcription from the book describing the transactions that compose the Saga:

> This Saga consists of the following local transactions:
> 1. **Order Service** — Create an Order in an APPROVAL_PENDING state
> 2. **Consumer Service** — Verify that the consumer can place an order
> 3. **Kitchen Service** — Validate order details and create a Ticket in the CREATE_PENDING state
> 4. **Accounting Service** — Authorize consumer's credit card
> 5. **Kitchen Service** — Change the state of the Ticket to AWAITING_ACCEPTANCE
> 6. **Order Service** — Change the state of the Order to APPROVED

In the book where this business scenario is presented, this Saga is based on the business logic of the application, which is presented in the chapters before. A detailed explanation of what the referred order and ticket states are is not required to understand the behavior of the framework. Observe that these six transactions mentioned in the transcription are represented in Figure 3.1. The first four transactions are the compensatable transactions in the figure, and the last two are the retriable transactions. The compensating transactions were approached in a further section in the book, and those cancel, in case of failure, the order creation and the ticket creation. These are represented in the figure as well.

As it was mentioned, a Saga is defined by a graph and the Saga precedence map. The above-mentioned figure represents the saga definition graph that was defined for this business case. The Saga precedence map definition is also required. As stated in Section 3.2.1, there exists an entry of the Saga precedence map for every transaction and end-state, and each entry is composed of an integer and a set of nested sets of task identifiers. As every task and end-state node only has one graph precedence relation, with the exception of the task 'REJECT_ORDER', only one set containing the respective precedent task is required for every node. Regarding the 'REJECT_ORDER' task, its execution can be triggered in two situations: after the failure of the task 'VERIFY_CUSTOMER' or after the success of the task 'REJECT_TICKET'. Therefore, one set has to be defined for each situation containing the identifier of the precedent task. The Redis commands of lines 17 to 38 of Listing A.1 define this precedence map of this Saga. The remaining commands of that listing define the graph presented in Figure 3.1.

4.2.2.2 Parallel Order Food Saga

In order to demonstrate the capabilities of the framework to handle parallel Sagas a second version of the order food Saga was created, emphasizing the parallel task execution as much as possible. The business logic of the tasks was neglected as the objective was to prove that the framework could execute Sagas with parallel tasks.

The Saga definition graph for the parallel version of the order food Saga is presented in Figure 4.1. As can be observed, in the Saga there are three parallel sections. The tasks 'VERIFY_CUSTOMER', 'CREATE_ORDER', and 'CREATE_TICKET' were set in parallel, as well as the 'APPROVE_TICKET' and 'APPROVE_ORDER' tasks and the 'REJECT_TICKET' and 'REJECT_ORDER' tasks.

As for the Saga precedence map, which is also part of the Saga definition, some changes also had to be made. The task 'AUTHORIZE_CARD' and both the end-states are now preceded by more than one task. In all three situations, it is required that all the graph precedent tasks have been executed for the framework to process the task or enter the end-state. Thus, the entries of the Saga precedence map for the 'AUTHORIZE_CARD' task and the end-states 'SUCCESS' and 'COMPENSATED' will contain only one set. For this reason, the Saga precedence map entry of every task and end-state will be composed of the graph precedent tasks.

Listing A.2 states the necessary commands to define the Saga using Redis commands.

4.2.2.3 Activate Prepaid SIM Card Saga

The second business scenario that will be used to validate the framework's PoC is an activation of a prepaid SIM card in the context of a telecommunications company. This scenario approximates a real-world scenario as described in an interview with seasoned experts in the field of integrating telecommunication systems from Deloitte (S. Lopes, B. Vasquez, and P. Branco, personal communication, June 3rd, 2023). The aim of the introduction of this scenario is to exemplify a use case in

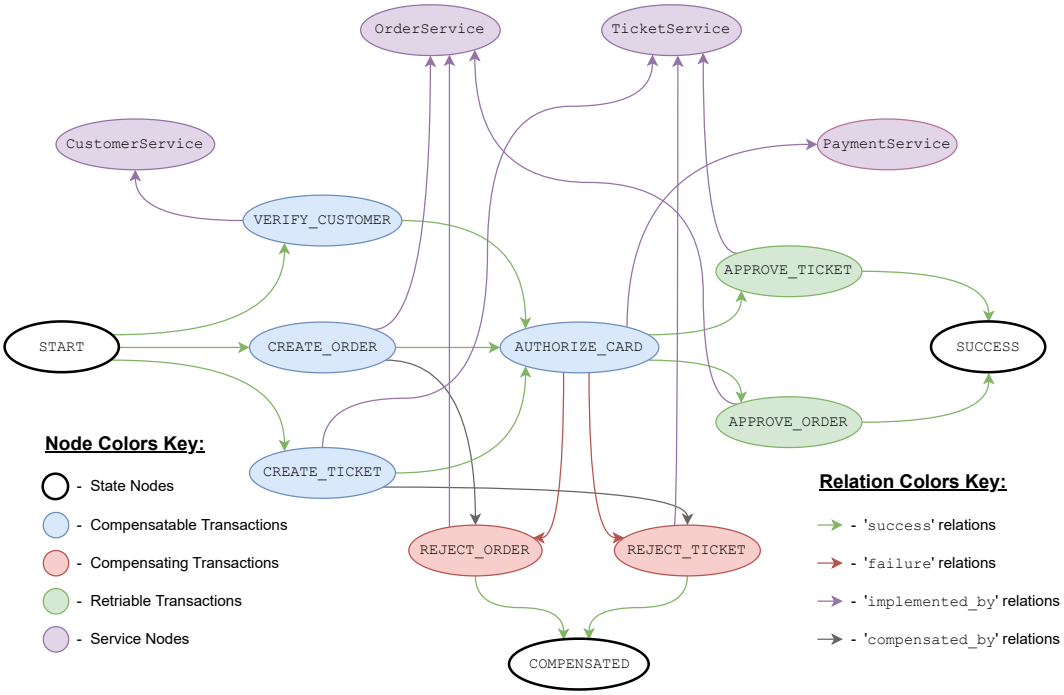


Figure 4.1: Parallel version of the order food Saga graph.

the field of integration where applying the framework proposed in this work would serve as a significant accelerator of the development process. Further details on the real-world case cannot be divulged due to Deloitte's pledged confidentiality to its clients. The Saga described in this section is a typical use case of the integration of telecommunications systems.

A telecommunications system is composed of various services with very distinct purposes. For the scope of this Saga, five services are going to be used: a number management system that manages the phone numbers associated with the telecommunications company; a billing service responsible for billing customers in the scope of their telecommunication contracts; a network service that, among other things, ensures that the network recognizes SIM cards by associating them with the respective phone number; the Enterprise Resource Planning (ERP) system that manages various business operations including inventory management, sales, and customer records; a communications service for sending out communications to customers via SMS.

According to the aforementioned interview (S. Lopes, B. Vasquez, and P. Branco, personal communication, June 3rd, 2023), a typical activation of a prepaid SIM card starts by reserving the phone number associated with the SIM card in the number management system. Having reserved the phone number, the billing service is invoked and a takeover of the contract associated with the prepaid card is tried with the customer's personal and billing information. This task can fail due to the customer not having a valid payment method, and, in that case, the phone number that was previously reserved in the number management system is freed. However, if the task succeeds, the SIM's phone number is updated in the network service, in order for the service to associate the phone number with the SIM card. At this point, the SIM card is ready to be utilized. The ERP

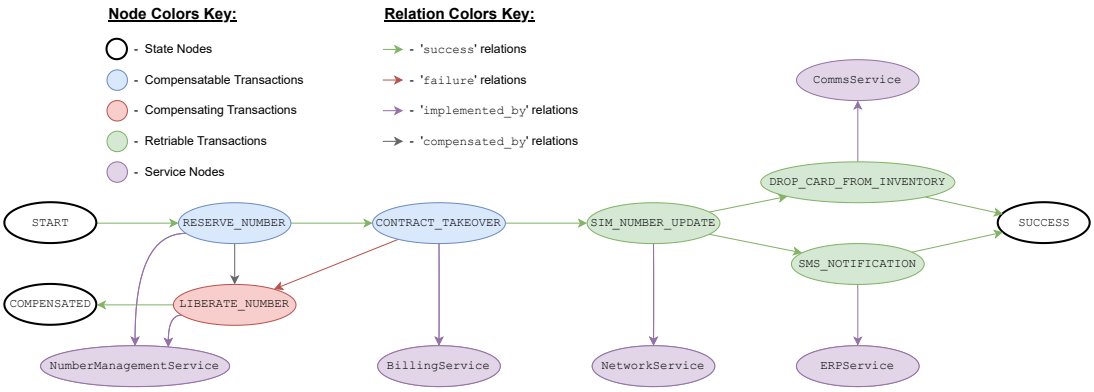


Figure 4.2: Activate prepaid SIM card Saga graph.

system must also be updated with the side effects of activating the SIM card, such as dropping the SIM card from the inventory and updating sales and customer data. Furthermore, it can also be interesting to notify the customer via SMS to their newly acquired SIM card's phone number that the card is ready to be utilized. This would be done using the communications service mentioned above. As Figure 4.2 shows, the last two tasks can be executed in parallel.

Figure 4.2 represents the graph that partially defines this Saga. As there are no situations in the Saga where a task can be executed (or an end-state can be entered) after the completion of two different tasks, the entries of every task and end-state in the Saga precedence are one set that contains the precedent tasks represented in the graph. Listing A.3 lists the Redis commands for defining this Saga.

4.2.2.4 Randomly Generated Saga

Following the rules for Saga definitions from Section 3.2.1, a random Saga generator was created. The Saga generator creates random Sagas with up to 40 tasks by generating the graph and the precedence map. Its main feature is that it generates parallel Sagas. The level of parallelism of generated Sagas is determined by two inputs to the generator, which are the probability of opening or closing a branch each time a new task is added to the generated Saga. When adding compensatable transactions, the probability of opening a branch was set to a higher value than the probability of closing one, and for the retriable transactions, the opposite approach was taken. Using this generator, it is possible to test whether the framework is able to execute very complex parallel Sagas, as complexity can be easily tuned. Figure 4.3 shows a randomly generated Saga graph.

The Saga generator has some limitations, though. The generated Saga precedence maps have little randomness. Every entry has one set only, and the set is created by consulting the generated Saga graph and adding all the precedent tasks to that set. This implies that the Task Execution Assertion mechanism presented in Section 3.3.1 will not be tested by executing random Sagas. More importantly, generated Sagas do not contain compensating transactions. This makes the

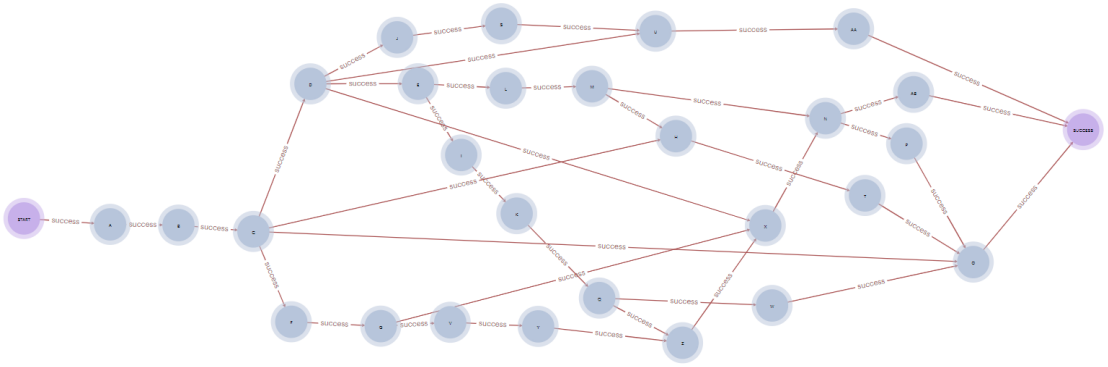


Figure 4.3: Randomly generated Saga graph. Only task and state nodes are shown for the sake of simplicity.

tests performed on generated Sagas not suitable for understanding if the framework behaves as expected for business failure situations. These limitations were set due to time constraints.

Nonetheless, using the Saga generator, it is still possible to validate whether, for business-wise successful situations, the framework delivers the execution of complex parallel Sagas. Four different generated Sagas were utilized.

4.2.3 Results

The proposed validation method presented in Section 4.2.1 was applied to the Sagas created in Section 4.2.2. In order to perform the validations, the running environment for each scenario was previously set up. A total of ten different scenarios were used for validation: each Saga, with the exception of the randomly generated Sagas, was executed in a situation where all the compensatable tasks succeeded, and in a situation where one of the compensatable tasks failed (business-wise). This led to two executions of the order food Saga, its parallel version, and the activate prepaid SIM card Saga. Adding the (always-successful) executions of the four different random Sagas, a total of ten validation situations were devised.

For performing the validation of the framework's behavior, the aspects presented in Section 4.2.1 were verified. Those verifications were done for each of the ten validation scenarios. Every verification was successful. Due to their extensiveness, the outputs of the Redis commands required to perform the validation are present in the appendix B.2.

4.3 Validation 2: Time Reduction of API Reuse

A second validation was done to verify whether this framework accomplishes its objective of reducing the time it takes to reutilize tasks that are implemented in APIs. This was done by measuring the required time for a MuleSoft developer to reutilize implemented tasks using the Shepherd framework, and by comparing this measurement to a baseline estimate of how long it takes to achieve the same goal without the Shepherd framework.

4.3.1 Scenario Description

Consider a successful telecommunications company that offers a variety of services, such as phone and internet services. One day, a business requirement for the introduction of prepaid SIM cards to its range of products arises, and a crucial step for its implementation is the development of the activation process of those SIM cards. Since the company already offers postpaid SIM cards, the tasks required to activate a prepaid SIM card are already implemented. However, the same process cannot be utilized as there are some tasks of the activation of the postpaid SIM cards that should not be executed in the activation of a prepaid SIM card. For this reason, a new process must be created for the activation of a prepaid SIM card where the tasks required to perform the novel activation are invoked.

Assume that the telecommunications company uses MuleSoft as the platform to build their backend systems, and because of that, an approach of API-led connectivity is followed. This means the system is built on several APIs, distributed on three layers of abstraction: the system, process, and experience layers. It is assumed that the tasks required to activate a prepaid SIM card are already implemented across APIs in the system layer.

As one might suspect, this scenario is a continuation of the one presented in Section 4.2.2.3. The tasks that compose the activation of a prepaid SIM card are presented there, along with the services that are part of this operation. Consider that the services presented in that section are integrated into the company's application network using APIs in MuleSoft.

4.3.2 Method

This validation compares two forms of creating a new application process for the activation of a prepaid SIM card. This comparison is accomplished by evaluating the execution times of these two forms. The traditional method for the creation of a new application process is the creation of a new process API that invokes the required tasks for performing the application process. This is the first form and will be referred to as the baseline method. The Shepherd framework enables an alternative form of creating application processes. Instead of developing a new API, the framework's user just requires to define a new Saga and make sure that the services which participate in that Saga are integrated into the framework. This is the second form and will be referred to as the Shepherd method.

The development time of the traditional method was estimated by interviewing the same quorum of seasoned integration experts that assisted in the creation of the scenario used for this validation (S. Lopes, B. Vasquez, and P. Branco, personal communication, June 24th, 2023). The quorum agreed on an estimate of two work days for an average MuleSoft developer to apply the baseline method. Based on the collective experience of the quorum's members, this estimate was proposed assuming that a new API is created that invokes the required tasks to implement a new application process and that the business logic of those tasks is already implemented. The process of creation of a new API is defined as its design, development, testing, and deployment. For these

reasons, the development time of the baseline method considered for this validation will be sixteen hours, assuming one workday is eight hours.

The development time of the Shepherd method was measured by having a MuleSoft developer[4] use the Shepherd framework to create the application process for activating a prepaid SIM card. The developer was given the challenge of defining and executing a Saga that invokes the required tasks to perform an activation of a prepaid SIM card. This involves integrating the services which implement those tasks into the framework[5]. Before the challenge, a one-hour presentation of the framework, the context of the experiment, and their challenge were given to the developer by the author of this work. Afterward, the developer's computer was set up for the execution of the experiment, which involved installing Docker and its dependencies. Only after the setup phase, which took about thirty minutes, the measurement began. The time counting finished when the developer was able to perform a successful Saga execution. A successful Saga execution is defined as a Saga execution whose Context enters the 'SUCCESS' state.

For the challenge, the developer was supplied with several resources. Firstly, a Saga definition graph was provided in a PDF format (figure 4.2), as a specification of the application process that should be built. Along with the graph, the Mule projects of the mentioned services were also made available, for them to be integrated into the framework. The criteria for the choice of the following resources was: what would be present in a Shepherd framework starting guide? Step-by-step instructions of what needs to be done in order to integrate each service into the framework are contained in the deliverable to the developer. These instructions also mentioned how to set up the Redis instance that implements the Context Manager. Furthermore, a Java project that implements the framework's logic, which should be packaged as a Maven dependency was provided. Also, the Interface, which is a Mule API, was contained in the deliverable to the developer. A Postman collection was shared with the developer in order for him to test his solution. Lastly, an example of another scenario where the framework was used to execute a Saga was shared.

4.3.3 Results

As mentioned before, a total of one measurement was made to the application of the Shepherd method. The developer who was subject to the experiment took 1:47:06 to solve the challenge after receiving the described briefing and setting up their working environment.

As it was described, the estimation of the duration of the baseline method was done by a quorum of seasoned integration experts (S. Lopes, B. Vasquez, and P. Branco, personal communication, June 24th, 2023). The quorum agreed that the development of the API described in the previous section would take two full workdays for an average MuleSoft Developer, where a full workday contains eight hours of work. Thus, a duration of 16 hours was considered for the application of the baseline method.

[4]The developer has been a MuleSoft practitioner for one year and possesses the certification "MuleSoft Certified Developer - Level 1".

[5]For clarification purposes, dummy tasks were used, i.e., Mule flows would be invoked that would just wait for an arbitrary time. The business logic of the tasks is not the object of study here

Due to time and resource restrictions, no more measurements of the proposed experiment were taken. Obviously, this reduces the scientific relevance of this validation method. However, these results were considered for two reasons: a significant improvement of the implementation time of the Shepherd method over the baseline method was predicted by the aforementioned quorum of integration experts; as the reduction of the development time is so accentuated (an efficiency[6] improvement of 896%), there is a large margin for measurement of errors in order for the improvement to still be significant.

4.4 Discussion

From the validation of the framework's behavior, it is possible to conclude with confidence that the Shepherd framework successfully implements the Saga Pattern using the choreography of microservices in a generic manner. All the verifications done to the framework execution's side effects were successful.

Observe that the definitions of microservices choreography [15, 12] state that there should not be a central component in a system that implements the choreography of microservices. Shepherd differs from the definition of pure choreography in that regard since it relies on the Context Manager as a shared memory between the microservices. It is also worth mentioning that this does not mean that the framework presented in this work leans more towards the orchestration of microservices instead of choreography, as the Context Manager is not an orchestrator; it does not define the flow of events, it just keeps the state of executions.

The validation of the framework's behavior could have been more robust. Specifically regarding unsuccessful random scenarios. One of the stated limitations of the Saga generator was that compensating tasks were not included in the generated Sagas. This decreases the robustness of the validation done to the framework. However, the framework's logic is very similar when it deals with unsuccessful scenarios. The only difference is that after a failing task, a different set of graph relations are used to determine what are the next tasks. As this is tested by three validation scenarios, it is considered not to be a hindrance to validating that the framework's behavior successfully implements the Saga pattern.

From the second validation, it is possible to conclude that the framework stimulates an improvement of 896% in the time it takes to reutilize implemented tasks in APIs. This value was obtained by comparing the result of the measurement of the Shepherd method's application (1:47:06) to the estimate of the baseline method's application (16:00:00). The stated percentage is the division of the latter value by the former value. Observe that the difference between the considered durations of the applications of both methods is 14:12:54.

This improvement is mainly due to Shepherd being a tool that requires only configuration and not necessarily development. The development of a new API includes time-consuming activities, such as design, testing, and deployment, whereas, for the users of Shepherd, those activities are

[6] Efficiency is defined as time dedicated to a task.

abstracted from them. As it was described in Section 3.5, the Shepherd framework user experience is based on a configuration-first approach, which aims for users solely to configure the framework and not build new applications or features. Moreover, observe that, as for any other tool, there is a learning curve to the usage of Shepherd. It is expected that by using the framework the developers will accelerate even more their experience with Shepherd.

Mind that the measured increase in productivity that this framework provides is only for task reutilization, which is one of many more jobs that compose the activities of a MuleSoft developer. Nonetheless, it still is a significant improvement in performance. Consider the above-mentioned difference between the considered durations of both methods: 14:12:54. If this reduction of development time is leveraged four times in a month, more than a work week of a professional was saved and reinvested into another activity. The reuse of tasks[7] is one of the backbones of approaches like API-led connectivity, which are fundamental to the field of integration. Therefore, situations, where development time could be reduced with the Shepherd framework, are recurrent. Furthermore, there is also the possibility of there being other performance gains that were not measured in this work.

How does the Shepherd framework fit in the API-led connectivity approach? The introduction of this framework to a MuleSoft system implies the introduction of a new process API to that system: the framework's Interface component. Observe that the Interface is a process API as it encapsulates specific business processes or services[8], which are defined by Sagas. However, the proposed framework does not offer API consumption capabilities, since its form of invoking tasks is by communicating with events through an event broker. For this reason, services and APIs must be integrated into the framework for their tasks to be invokable by Sagas. This clearly deviates from the API-led connectivity principles, which state that the communication between the applications that compose a system should be made via the consumption of APIs. Furthermore, it is important to remark that the design of the framework does not allow for Sagas to invoke other Sagas. This, however, is one of the most important features of API-led connectivity: APIs can invoke other APIs to form more complex business logic.

[7] Or assets. An implemented task is a digital asset.

[8] This definition was taken from Section 2.3.1

Chapter 5

Conclusions

This thesis's purpose was to study the viability of implementing the Saga Pattern using the choreography of microservices in a generic manner in MuleSoft. In order to achieve its broader purpose, at the beginning of this work, four objectives were presented. Firstly, this work should have proposed a MuleSoft framework that implements the Saga Pattern regardless of the business use case using the choreography of microservices. In order for it to be applicable in the industry, the proposed framework should blend with state-of-the-art concepts of the field of integration, such as API-led connectivity. Furthermore, the process of reutilizing APIs in an API-led environment should be accelerated by using a configuration-first approach to the development of the framework's user experience.

Following the delineated objectives, the Shepherd framework was proposed in this work. As it was discussed in section 4.4, this dissertation concludes that the proposed framework implements the Saga Pattern, based on the assumption that the validation performed to the framework's behavior[1] is valid. Furthermore, it was concluded that the process of reutilization of APIs described in section 4.3 was accelerated by 896%. The main cause of this improvement was attributed to the configuration-first approach of the framework usage experience. Lastly, it was also concluded in the above-mentioned section that the usage of the framework has some impacts on the API-led connectivity strategy.

The acceleration mentioned above translates into an increase in developers' productivity. An increase in developers' productivity is desirable for every organization, as well as for the developers themselves. It allows for projects to be completed more quickly; this could mean that new products, updates, or features can be released more rapidly, giving an organization such as Deloitte a competitive edge in the market. The primary goal of most businesses is to increase profits; increasing productivity could potentially lead to cost savings, as projects can be completed faster with fewer resources, and these savings can directly increase profits. Ultimately, Deloitte, with whom this work was developed, could potentially take on more projects without having to proportionally increase its workforce, and this would mean that it could (potentially) grow and scale its business more efficiently.

[1] See the section 4.2.1

The framework proposed in this work challenges the API-led way of orchestrating tasks. In the API-led connectivity approach, creating an orchestration of tasks is accomplished by developing a process or experience API that invokes other APIs. With the Shepherd framework, orchestrating (or *choreographying*) a set of tasks is done by defining a Saga, i.e., configuring Shepherd to execute a set of tasks in a determined sequence. Therefore, this framework enables a configuration-first approach to orchestrating the execution of tasks, which differs from the current form of task orchestration in an API-led environment.

The introduction of the Shepherd framework to an API-led environment can be seen as the introduction of a configurable process API, which is the framework's Interface. The framework's Interface exposes the capability of executing Sagas to other APIs. Process APIs are defined as APIs that "encapsulate specific business processes or services" (see section 2.3.1). This description matches the framework's Interface purpose; it is encapsulating the execution of Sagas, and a Saga is nothing more than a business process. This is a significant contribution to the field of integration: Shepherd creates a configurable process API.

Besides the impacts on API-led environments, an alternative point of view is that Shepherd extends the integration patterns that MuleSoft supports: as it was its purpose, with Shepherd, the Saga Pattern becomes one more pattern to use to build integrations in MuleSoft. Sagas were originally proposed just as a transaction management solution, out of the context of microservices or systems integration. In spite of today's trends of adopting the Saga Pattern in microservices architectures, it is still a technique that can be applied in other situations such as building with integrations with MuleSoft.

For all the reasons mentioned in this section, Shepherd is a framework with a high potential for application in the industry and for being subjected to further research and development. The following section presents suggestions for future research directions.

5.1 Future Work

The first next step to take in the development of the Shepherd framework is to lift the limitations imposed on the PoC that were mentioned in section 4.1.1. This means adding the return values mechanism to the framework's implementation, testing the framework in an environment that is more similar to a real-world scenario, and enabling the parallel execution of tasks in each API (or microservice). The first constraint can be addressed by following the description of the mechanism present in section 3.3.3. Lifting the second restriction is addressed in the following paragraph. The last limitation can be lifted by treating the Asserter component as a critical section, and ensuring that only one task per Context and service executes the Asserter component at the same time.

Several issues must be tackled first before subjecting Shepherd to a real-world scenario. A testing environment must be created that better simulates a real-world scenario. This will identify challenges required to be solved in order to make the framework industry-ready. In a real-world distributed system, issues such as concurrency, scalability, and technical failures must be handled

in a robust manner. These issues were not addressed in this work. In order to address them, more mechanisms such as the ones presented in Section 3.3 must be developed or the technology choices must be adapted. For example, the Context Manager was implemented as a single instance of Redis, the default Redis deployment topology. This is not a scalable or highly available solution. Technologies such as Dynomite [41] should be considered for transforming the Context Manager into a production-ready component. Dynomite offers linear scalability, high availability, and other desirable properties in a production-ready component. Additionally, it should be studied whether locking mechanisms are required when performing changes to Contexts so that Contexts are not left in inconsistent states.

Regarding the event broker, the technology which implements it should also be reconsidered. According to a conversation with the aforementioned quorum of seasoned integration experts (S. Lopes, B. Vasquez, and P. Branco, personal communication, May 15th, 2023), it is desirable to minimize the technology stack of systems that are deployed to production. This increases the maintainability, simplicity, and potentially decreases the cost of the system. A proposal of two different options that would enhance the event broker follows: firstly, Redis can be used as the event broker. This would reduce the technology stack. Redis has two different possibilities of implementing event brokers: Redis Pub/Sub and Redis Streams. Only Redis Streams can be considered to replace the current event broker, as Redis Pub/Sub only offers the event delivery guarantee *at-most-once*. The second option would be to use an event broker that enables the use of event sourcing, such as Apache Kafka. In order for event sourcing to be supported by an event broker, it needs to persist all the events it receives, even after the events are sent to their destination. By doing so, in case of a failure to the Context Manager, it is possible to reconstruct the system's state by replaying the persisted events.

Saga definitions have space for improvement as well. It would be interesting to add the feature of having conditional branches in Sagas. With conditional branches, it is meant that depending on the output of a certain task, different sequences of activities could follow. Another interesting further development possibility would be to create a tool that would determine if a Saga definition is valid. This would make the framework more user-friendly and would be a significant step to making it ready to be applied in the industry.

A possibility for future development of the framework is to make it fully compliant with the API-led connectivity approach. The main deviation of Shepherd from the API-led principles is that it is not able to consume HTTP APIs. In the framework, task executions are triggered by events sent to microservices (or APIs) via a connection to an event broker. This leads to having to create a connection to the event broker in each API so that their tasks are invokable by the framework. In order to remove this necessity and have the framework integrate an API-led environment without having to perform changes to APIs, the framework's logic would have to be moved to a central orchestrator. This central orchestrator would be able to consume APIs via HTTP and would invoke the Sagas' tasks by sending HTTP requests. This is the only solution to make the framework not require changes to APIs, since the distributed architecture presented in section 3.4 distributes the computational load of the framework's logic throughout the participants of the framework. The

architectural components presented in the aforementioned section, the Expecter, the Asserter, and the Router, would be executed by the central orchestrator. Therefore, the central orchestrator would replace the event broker. Changing the framework's architecture to a centralized approach in order to comply with the API-led connectivity principles would have significant implications and is a solution that requires further investigation.

Lastly, in order to increase the confidence of the results of the validation presented in Section 4.3, more measurements of the required time for a MuleSoft developer to reutilize tasks using Shepherd should be carried out. This experiment was presented in the mentioned section, and it would only be necessary to replicate it with more MuleSoft developers. Repeating the experiment will contribute to the scientific robustness of this work.

5.2 Final Statements

This dissertation has fulfilled its purpose: the Shepherd framework implements the Saga Pattern using the choreography of microservices, regardless of the business context, as it was expected. Also, the specific objectives mentioned at the beginning of this work, and reminisced at the beginning of this section were also accomplished. Nonetheless, there is always room for improvement, and future research and development directions were suggested.

Besides the framework as a new product, this work's main contribution to the field was the development of a process API with configurable business logic. The creation of this concept was a side effect of the development of the Shepherd framework, and as it was seen, it has a high potential of increasing MuleSoft developers' productivity.

A scientific article on this work is going to be submitted to a scientific conference. Its already-submitted abstract is present in appendix C.

References

[1] Deloitte Global, "Deloitte | Audit, Consulting, Financial, Risk Management, Tax Services," https://www.deloitte.com/global/en.html, [accessed 06/29/2023].

[2] Mulesoft, "Deloitte Digital," https://www.mulesoft.com/partner/deloitte-digital, [accessed 07/05/2023].

[3] H. Garcia-Molina and K. Salem, "Sagas," *ACM SIGMOD Record*, vol. 16, pp. 249–259, 1987.

[4] E. Daraghmi, C.-P. Zhang, and S.-M. Yuan, "Enhancing saga pattern for distributed transactions within a microservices architecture," *Applied Sciences*, vol. 12, p. 6242, 6 2022.

[5] G. Vial, "Understanding digital transformation: A review and a research agenda," *The Journal of Strategic Information Systems*, vol. 28, no. 2, pp. 118–144, 2019, sI: Review issue. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0963868717302196

[6] F. G. A. Pires, "Decision Support Tool for Integration Platforms," Master's thesis, Facludade de Engenharia da Universidade do Porto, Porto, 2020.

[7] Enterprisers Project, "What is digital transformation? Your top questions answered," 2020, Whitepaper, https://enterprisersproject.com/what-is-digital-transformation, [accessed 03/26/2023].

[8] F. Contreras, E. Baykal, and G. Abid, "E-Leadership and Teleworking in Times of COVID-19 and Beyond: What We Know and Where Do We Go," *Frontiers in Psychology*, 2020. [Online]. Available: www.frontiersin.org

[9] B. van Gils and H. A. Proper, "Enterprise modelling in the age of digital transformation," *Lecture Notes in Business Information Processing*, vol. 335, pp. 257–273, 2018. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-02302-7_16

[10] Mulesoft, "What is integration?" https://www.mulesoft.com/resources/integration, [accessed 04/03/2023].

[11] IBM Cloud Education, "Enterprise Integration: What It Is and Why It's Important," 6 2021, https://www.ibm.com/cloud/blog/enterprise-integration, [accessed 04/03/2023].

[12] C. Richardson, *Microservices Patterns: With examples in Java*. Shelter Island, NY: Manning Publications Co., 2019.

[13] A. Megargel, C. M. Poskitt, and V. Shankararaman, "Microservices Orchestration vs. Choreography: A Decision Framework," *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 134–141, 2021.

[14] D. Namiot and M. Sneps-Sneppe, "On Micro-services Architecture," *International Journal of Open Information Technologies*, vol. 2, 2014.

[15] M. Richards and N. Ford, *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, Inc., 2020.

[16] C. Richardson, "Database per service," https://microservices.io/patterns/data/database-per-service.html, [accessed 06/08/2023].

[17] Netflix Inc., "Conductor Documentation," https://conductor.netflix.com/, [accessed 03/26/2023].

[18] Uber Technologies Inc., "Cadence," https://cadenceworkflow.io/, [accessed 03/26/2023].

[19] N. Singhal, U. Sakthivel, and P. Raj, "Selection Mechanism of Micro-Services Orchestration Vs. Choreography," *International journal of Web & Semantic Technology*, vol. 10, pp. 01–13, 1 2019.

[20] M. H. Ali, R. Hasan, and M. Benyoucef, "Measuring the Modeling Complexity of Microservice Choreography and Orchestration: The Case of E-Commerce Applications," *SSRN Electronic Journal*, 2022.

[21] C. K. Rudrabhatla, "Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture," *International Journal of Advanced Computer Science and Applications*, vol. 9, 2018.

[22] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture," *ACM SIGAPP Applied Computing Review*, vol. 17, pp. 29–45, 1 2018.

[23] X. Bai, I. Arapakis, B. B. Cambazoglu, and A. Freire, "Understanding and Leveraging the Impact of Response Latency on User Behaviour in Web Search," *ACM Transactions on Information Systems*, vol. 36, pp. 1–42, 4 2018.

[24] J.-Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon, "Isotope: ACID Transactions for Block Storage," *ACM Transactions on Storage*, vol. 13, pp. 1–25, 2 2017.

[25] E. Brewer, "Cap twelve years later: How the "rules" have changed," *Computer*, vol. 45, pp. 23–29, 2 2012.

[26] G. Speegle, "Compensating Transactions," *Encyclopedia of Database Systems*, pp. 405–406, 2009.

[27] U. Friedrichsen, "The limits of the Saga pattern," 2 2021, https://www.ufried.com/blog/limits_of_saga_pattern/, [accessed 06/09/2023].

[28] Salesforce, "What Is MuleSoft? What Does MuleSoft Do?" https://www.salesforce.com/blog/what-is-mulesoft/, [accessed 04/14/2023].

[29] MuleSoft, "Anypoint Design Center | Design and Build Integrations," https://www.mulesoft.com/platform/anypoint-design-center, [accessed 04/13/2023].

[30] ——, "Anypoint Studio | Integrated Development Environment (IDE)," https://www.mulesoft.com/platform/studio, [accessed 04/13/2023].

[31] ——, "Anypoint Exchange | Design and Build Integrations," https://www.mulesoft.com/platform/exchange, [accessed 04/13/2023].

[32] ——, "Anypoint Management Center | Full Lifecycle API Management," https://www.mulesoft.com/platform/anypoint-management-center, [accessed 04/13/2023].

[33] ——, "Tutorial: Build an API from Start to Finish," https://www.mulesoft.com/platform/anypoint-design-center, [accessed 04/14/2023].

[34] K. Sharma, Kshamta, and A. Joshi, "API LED Connectivity Approach," *2022 International Conference on Cyber Resilience (ICCR)*, pp. 1–3, 10 2022.

[35] Mulesoft, "API-led Connectivity," 2023, Whitepaper, https://www.mulesoft.com/lp/whitepaper/api/api-led-connectivity, [accessed 04/10/2023].

[36] ——, "MuleSoft Accelerator for Life Sciences," https://www.mulesoft.com/exchange/org.mule.examples/mulesoft-accelerator-for-life-sciences/minor/1.1/, [accessed 04/14/2023].

[37] P. Choudhary, "What Is API-led Connectivity? Unlock Business Agility," https://blogs.mulesoft.com/learn-apis/api-led-connectivity/what-is-api-led-connectivity/, [accessed 05/27/2023].

[38] Mulesoft, "Mule Overview | MuleSoft Documentation," https://docs.mulesoft.com/mule-runtime/4.4/, [accessed 04/14/2023].

[39] MuleSoft, "Step 3. Develop the API," https://docs.mulesoft.com/general/api-led-develop, [accessed 06/29/2023].

[40] M. Wahnon, "awesome-workflow-engines," GitHub, https://github.com/meirwah/awesome-workflow-engines, [accessed 07/09/2023].

[41] Netflix Inc., "Netflix Dynomite Wiki," https://github.com/Netflix/dynomite/wiki, [accessed 06/29/2023].

[42] International Conference on Service-Oriented Computing, "Home - ICSOC 2023," https://icsoc2023.diag.uniroma1.it/, [accessed 07/02/2023].

Appendix A

Saga Definition Examples

A.1 Order Food Saga

Listing A.1 is an example of a saga definition in Redis for the framework's proof of concept. The saga serves as a fictitious example of the framework being applied to a food delivery system where an order is performed. This context was taken from [12].

```
1  GRAPH.QUERY Saga:order_food "CREATE (:State {name: 'START'})"
2  GRAPH.QUERY Saga:order_food "CREATE (:State {name: 'SUCCESS'})"
3  GRAPH.QUERY Saga:order_food "CREATE (:State {name: 'COMPENSATED'})"
4  GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'CREATE_ORDER', type: '
       COMPENSATABLE'})"
5  GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'VERIFY_CUSTOMER', type: '
       COMPENSATABLE'})"
6  GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'CREATE_TICKET', type: '
       COMPENSATABLE'})"
7  GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'AUTHORIZE_CARD', type: '
       COMPENSATABLE'})"
8  GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'APPROVE_TICKET', type: '
       RETRIABLE'})"
9  GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'APPROVE_ORDER', type: 'RETRIABLE
       '})"
10 GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'REJECT_ORDER', type: '
       COMPENSATING'})"
11 GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'REJECT_TICKET', type: '
       COMPENSATING'})"
12 GRAPH.QUERY Saga:order_food "CREATE (:Service {name: 'OrderService', address: '
       joaocapinto-tese-order-queue'})"
13 GRAPH.QUERY Saga:order_food "CREATE (:Service {name: 'CustomerService', address: '
       joaocapinto-tese-customer-queue'})"
14 GRAPH.QUERY Saga:order_food "CREATE (:Service {name: 'TicketService', address: '
       joaocapinto-tese-ticket-queue'})"
15 GRAPH.QUERY Saga:order_food "CREATE (:Service {name: 'PaymentService', address: '
       joaocapinto-tese-payment-queue'})"
```

```
16
17 SET Saga:order_food:Task:CREATE_ORDER:RequiredTasksSets 1
18 SADD Saga:order_food:Task:CREATE_ORDER:RequiredTasksSet:0 "START"
19 SET Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSets 1
20 SADD Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSet:0 "CREATE_ORDER"
21 SET Saga:order_food:Task:CREATE_TICKET:RequiredTasksSets 1
22 SADD Saga:order_food:Task:CREATE_TICKET:RequiredTasksSet:0 "VERIFY_CUSTOMER"
23 SET Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSets 1
24 SADD Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSet:0 "CREATE_TICKET"
25 SET Saga:order_food:Task:APPROVE_TICKET:RequiredTasksSets 1
26 SADD Saga:order_food:Task:APPROVE_TICKET:RequiredTasksSet:0 "AUTHORIZE_CARD"
27 SET Saga:order_food:Task:APPROVE_ORDER:RequiredTasksSets 1
28 SADD Saga:order_food:Task:APPROVE_ORDER:RequiredTasksSet:0 "APPROVE_TICKET"
29 SET Saga:order_food:Task:REJECT_TICKET:RequiredTasksSets 1
30 SADD Saga:order_food:Task:REJECT_TICKET:RequiredTasksSet:0 "AUTHORIZE_CARD"
31 SET Saga:order_food:Task:REJECT_ORDER:RequiredTasksSets 2
32 SADD Saga:order_food:Task:REJECT_ORDER:RequiredTasksSet:0 "VERIFY_CUSTOMER"
33 SADD Saga:order_food:Task:REJECT_ORDER:RequiredTasksSet:1 "REJECT_TICKET"
34
35 SET Saga:order_food:State:SUCCESS:RequiredTasksSets 1
36 SADD Saga:order_food:State:SUCCESS:RequiredTasksSet:0 "APPROVE_ORDER"
37 SET Saga:order_food:State:COMPENSATED:RequiredTasksSets 1
38 SADD Saga:order_food:State:COMPENSATED:RequiredTasksSet:0 "REJECT_ORDER"
39
40 GRAPH.QUERY Saga:order_food "MATCH (t1:State {name: 'START'}), (t2:Task {name: '
       CREATE_ORDER'}) CREATE (t1)-[:success]->(t2)"
41 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'CREATE_ORDER'}), (t2:Task {name
       : 'VERIFY_CUSTOMER'}) CREATE (t1)-[:success]->(t2)"
42 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'VERIFY_CUSTOMER'}), (t2:Task {
       name: 'CREATE_TICKET'}) CREATE (t1)-[:success]->(t2)"
43 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'CREATE_TICKET'}), (t2:Task {
       name: 'AUTHORIZE_CARD'}) CREATE (t1)-[:success]->(t2)"
44 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'AUTHORIZE_CARD'}), (t2:Task {
       name: 'APPROVE_TICKET'}) CREATE (t1)-[:success]->(t2)"
45 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'APPROVE_TICKET'}), (t2:Task {
       name: 'APPROVE_ORDER'}) CREATE (t1)-[:success]->(t2)"
46 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'APPROVE_ORDER'}), (t2:State {
       name: 'SUCCESS'}) CREATE (t1)-[:success]->(t2)"
47 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'CREATE_ORDER'}), (t2:Task {name
       : 'REJECT_ORDER'}) CREATE (t1)-[:compensated_by]->(t2)"
48 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'CREATE_TICKET'}), (t2:Task {
       name: 'REJECT_TICKET'}) CREATE (t1)-[:compensated_by]->(t2)"
49 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'AUTHORIZE_CARD'}), (t2:Task {
       name: 'REJECT_TICKET'}) CREATE (t1)-[:failure]->(t2)"
50 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'VERIFY_CUSTOMER'}), (t2:Task {
       name: 'REJECT_ORDER'}) CREATE (t1)-[:failure]->(t2)"
51 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'REJECT_TICKET'}), (t2:Task {
       name: 'REJECT_ORDER'}) CREATE (t1)-[:success]->(t2)"
```

```
52 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'REJECT_ORDER'}), (t2:State {
       name: 'COMPENSATED'}) CREATE (t1)-[:success]->(t2)"
53
54 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'CREATE_ORDER'}), (s:Service {
       name: 'OrderService'}) CREATE (t)-[:implemented_by]->(s)"
55 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'REJECT_ORDER'}), (s:Service {
       name: 'OrderService'}) CREATE (t)-[:implemented_by]->(s)"
56 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'APPROVE_ORDER'}), (s:Service {
       name: 'OrderService'}) CREATE (t)-[:implemented_by]->(s)"
57 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'VERIFY_CUSTOMER'}), (s:Service {
       name: 'CustomerService'}) CREATE (t)-[:implemented_by]->(s)"
58 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'CREATE_TICKET'}), (s:Service {
       name: 'TicketService'}) CREATE (t)-[:implemented_by]->(s)"
59 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'REJECT_TICKET'}), (s:Service {
       name: 'TicketService'}) CREATE (t)-[:implemented_by]->(s)"
60 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'APPROVE_TICKET'}), (s:Service {
       name: 'TicketService'}) CREATE (t)-[:implemented_by]->(s)"
61 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'AUTHORIZE_CARD'}), (s:Service {
       name: 'PaymentService'}) CREATE (t)-[:implemented_by]->(s)"
```

Listing A.1: Example of a saga definition for food ordering.

A.1.1 Parallel Version

The following are the Redis commands to define the parallel version of the order food saga.

```
1  GRAPH.QUERY Saga:order_food "CREATE (:State {name: 'START'})"
2  GRAPH.QUERY Saga:order_food "CREATE (:State {name: 'SUCCESS'})"
3  GRAPH.QUERY Saga:order_food "CREATE (:State {name: 'COMPENSATED'})"
4  GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'CREATE_ORDER', type: '
       COMPENSATABLE'})"
5  GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'VERIFY_CUSTOMER', type: '
       COMPENSATABLE'})"
6  GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'CREATE_TICKET', type: '
       COMPENSATABLE'})"
7  GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'AUTHORIZE_CARD', type: '
       COMPENSATABLE'})"
8  GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'APPROVE_TICKET', type: '
       RETRIABLE'})"
9  GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'APPROVE_ORDER', type: 'RETRIABLE
       '})"
10 GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'REJECT_ORDER', type: '
       COMPENSATING'})"
11 GRAPH.QUERY Saga:order_food "CREATE (:Task {name: 'REJECT_TICKET', type: '
       COMPENSATING'})"
12 GRAPH.QUERY Saga:order_food "CREATE (:Service {name: 'OrderService', address: '
       joaocapinto-tese-order-queue'})"
```

```
13 GRAPH.QUERY Saga:order_food "CREATE (:Service {name: 'CustomerService', address: '
       joaocapinto-tese-customer-queue'})"
14 GRAPH.QUERY Saga:order_food "CREATE (:Service {name: 'TicketService', address: '
       joaocapinto-tese-ticket-queue'})"
15 GRAPH.QUERY Saga:order_food "CREATE (:Service {name: 'PaymentService', address: '
       joaocapinto-tese-payment-queue'})"
16
17 SET Task:CREATE_ORDER:RequiredTasksSets 1
18 SADD Task:CREATE_ORDER:RequiredTasksSet:0 "START"
19 SET Task:VERIFY_CUSTOMER:RequiredTasksSets 1
20 SADD Task:VERIFY_CUSTOMER:RequiredTasksSet:0 "START"
21 SET Task:CREATE_TICKET:RequiredTasksSets 1
22 SADD Task:CREATE_TICKET:RequiredTasksSet:0 "START"
23 SET Task:AUTHORIZE_CARD:RequiredTasksSets 1
24 SADD Task:AUTHORIZE_CARD:RequiredTasksSet:0 "CREATE_ORDER"
25 SADD Task:AUTHORIZE_CARD:RequiredTasksSet:0 "VERIFY_CUSTOMER"
26 SADD Task:AUTHORIZE_CARD:RequiredTasksSet:0 "CREATE_TICKET"
27 SET Task:APPROVE_TICKET:RequiredTasksSets 1
28 SADD Task:APPROVE_TICKET:RequiredTasksSet:0 "AUTHORIZE_CARD"
29 SET Task:APPROVE_ORDER:RequiredTasksSets 1
30 SADD Task:APPROVE_ORDER:RequiredTasksSet:0 "AUTHORIZE_CARD"
31 SET Task:REJECT_TICKET:RequiredTasksSets 1
32 SADD Task:REJECT_TICKET:RequiredTasksSet:0 "AUTHORIZE_CARD"
33 SET Task:REJECT_ORDER:RequiredTasksSets 1
34 SADD Task:REJECT_ORDER:RequiredTasksSet:0 "AUTHORIZE_CARD"
35
36 SET Saga:order_food:State:SUCCESS:RequiredTasksSets 1
37 SADD Saga:order_food:State:SUCCESS:RequiredTasksSet:0 "APPROVE_ORDER"
38 SADD Saga:order_food:State:SUCCESS:RequiredTasksSet:0 "APPROVE_TICKET"
39 SET Saga:order_food:State:COMPENSATED:RequiredTasksSets 1
40 SADD Saga:order_food:State:COMPENSATED:RequiredTasksSet:0 "REJECT_ORDER"
41 SADD Saga:order_food:State:COMPENSATED:RequiredTasksSet:0 "REJECT_TICKET"
42
43 GRAPH.QUERY Saga:order_food "MATCH (t1:State {name: 'START'}), (t2:Task {name: '
       CREATE_ORDER'}) CREATE (t1)-[:success]->(t2)"
44 GRAPH.QUERY Saga:order_food "MATCH (t1:State {name: 'START'}), (t2:Task {name: '
       VERIFY_CUSTOMER'}) CREATE (t1)-[:success]->(t2)"
45 GRAPH.QUERY Saga:order_food "MATCH (t1:State {name: 'START'}), (t2:Task {name: '
       CREATE_TICKET'}) CREATE (t1)-[:success]->(t2)"
46 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'CREATE_ORDER'}), (t2:Task {name
       : 'AUTHORIZE_CARD'}) CREATE (t1)-[:success]->(t2)"
47 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'VERIFY_CUSTOMER'}), (t2:Task {
       name: 'AUTHORIZE_CARD'}) CREATE (t1)-[:success]->(t2)"
48 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'CREATE_TICKET'}), (t2:Task {
       name: 'AUTHORIZE_CARD'}) CREATE (t1)-[:success]->(t2)"
49 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'AUTHORIZE_CARD'}), (t2:Task {
       name: 'APPROVE_TICKET'}) CREATE (t1)-[:success]->(t2)"
50 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'AUTHORIZE_CARD'}), (t2:Task {
       name: 'APPROVE_ORDER'}) CREATE (t1)-[:success]->(t2)"
```

```
51 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'APPROVE_TICKET'}), (t2:State {
       name: 'SUCCESS'}) CREATE (t1)-[:success]->(t2)"
52 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'APPROVE_ORDER'}), (t2:State {
       name: 'SUCCESS'}) CREATE (t1)-[:success]->(t2)"
53
54 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'CREATE_ORDER'}), (t2:Task {name
       : 'REJECT_ORDER'}) CREATE (t1)-[:compensated_by]->(t2)"
55 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'CREATE_TICKET'}), (t2:Task {
       name: 'REJECT_TICKET'}) CREATE (t1)-[:compensated_by]->(t2)"
56
57 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'AUTHORIZE_CARD'}), (t2:Task {
       name: 'REJECT_TICKET'}) CREATE (t1)-[:failure]->(t2)"
58 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'AUTHORIZE_CARD'}), (t2:Task {
       name: 'REJECT_ORDER'}) CREATE (t1)-[:failure]->(t2)"
59
60 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'REJECT_TICKET'}), (t2:State {
       name: 'COMPENSATED'}) CREATE (t1)-[:success]->(t2)"
61 GRAPH.QUERY Saga:order_food "MATCH (t1:Task {name: 'REJECT_ORDER'}), (t2:State {
       name: 'COMPENSATED'}) CREATE (t1)-[:success]->(t2)"
62
63 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'CREATE_ORDER'}), (s:Service {
       name: 'OrderService'}) CREATE (t)-[:implemented_by]->(s)"
64 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'REJECT_ORDER'}), (s:Service {
       name: 'OrderService'}) CREATE (t)-[:implemented_by]->(s)"
65 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'APPROVE_ORDER'}), (s:Service {
       name: 'OrderService'}) CREATE (t)-[:implemented_by]->(s)"
66 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'VERIFY_CUSTOMER'}), (s:Service {
       name: 'CustomerService'}) CREATE (t)-[:implemented_by]->(s)"
67 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'CREATE_TICKET'}), (s:Service {
       name: 'TicketService'}) CREATE (t)-[:implemented_by]->(s)"
68 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'REJECT_TICKET'}), (s:Service {
       name: 'TicketService'}) CREATE (t)-[:implemented_by]->(s)"
69 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'APPROVE_TICKET'}), (s:Service {
       name: 'TicketService'}) CREATE (t)-[:implemented_by]->(s)"
70 GRAPH.QUERY Saga:order_food "MATCH (t:Task {name: 'AUTHORIZE_CARD'}), (s:Service {
       name: 'PaymentService'}) CREATE (t)-[:implemented_by]->(s)"
```

Listing A.2: Example of a saga definition for food ordering (in parallel).

A.2 Activate Prepaid SIM Card Saga

Listing A.3 is an example of a saga definition in Redis for the framework's proof of concept. The saga serves as a fictitious example of the framework being applied to a telecommunications system where the activation of a SIM card is performed. This example was created in collaboration with seasoned industry experts (S. Lopes, B. Vasquez, and P. Branco, personal communication, June 2nd, 2023).

```
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:State {name: 'START'})"
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:State {name: 'SUCCESS'})"
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:State {name: 'COMPENSATED'})"
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:Task {name: 'RESERVE_NUMBER', type: 'COMPENSATABLE'})"
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:Task {name: 'CONTRACT_TAKEOVER', type: 'COMPENSATABLE'})"
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:Task {name: 'SIM_CARD_NUMBER_UPDATE', type: 'RETRIABLE'})"
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:Task {name: 'SMS_NOTIFICATION', type: 'RETRIABLE'})"
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:Task {name: 'DROP_CARD_FROM_INVENTORY', type: 'RETRIABLE'})"
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:Task {name: 'LIBERATE_NUMBER', type: 'COMPENSATING'})"
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:Service {name: 'NumberManagementService', address:'joaocapinto-tese-nms-queue'})"
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:Service {name: 'BillingService', address:'joaocapinto-tese-bill-queue'})"
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:Service {name: 'NetworkService', address:'joaocapinto-tese-net-queue'})"
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:Service {name: 'CommunicationsService', address:'joaocapinto-tese-comms-queue'})"
GRAPH.QUERY Saga:activate_prepaid_sim \"CREATE (:Service {name: 'ERPConnectionService', address:'joaocapinto-tese-erp-queue'})"

SET Saga:activate_prepaid_sim:Task:RESERVE_NUMBER:RequiredTasksSets 1
SADD Saga:activate_prepaid_sim:Task:RESERVE_NUMBER:RequiredTasksSet:0 "START"
SET Saga:activate_prepaid_sim:Task:CONTRACT_TAKEOVER:RequiredTasksSets 1
SADD Saga:activate_prepaid_sim:Task:CONTRACT_TAKEOVER:RequiredTasksSet:0 "RESERVE_NUMBER"
SET Saga:activate_prepaid_sim:Task:SIM_CARD_NUMBER_UPDATE:RequiredTasksSets 1
SADD Saga:activate_prepaid_sim:Task:SIM_CARD_NUMBER_UPDATE:RequiredTasksSet:0 "CONTRACT_TAKEOVER"
SET Saga:activate_prepaid_sim:Task:SMS_NOTIFICATION:RequiredTasksSets 1
SADD Saga:activate_prepaid_sim:Task:SMS_NOTIFICATION:RequiredTasksSet:0 "SIM_CARD_NUMBER_UPDATE"
SET Saga:activate_prepaid_sim:Task:DROP_CARD_FROM_INVENTORY:RequiredTasksSets 1
SADD Saga:activate_prepaid_sim:Task:DROP_CARD_FROM_INVENTORY:RequiredTasksSet:0 "SIM_CARD_NUMBER_UPDATE"
SET Saga:activate_prepaid_sim:Task:LIBERATE_NUMBER:RequiredTasksSets 1
SADD Saga:activate_prepaid_sim:Task:LIBERATE_NUMBER:RequiredTasksSet:0 "CONTRACT_TAKEOVER"

SET Saga:activate_prepaid_sim:State:SUCCESS:RequiredTasksSets 1"
SADD Saga:activate_prepaid_sim:State:SUCCESS:RequiredTasksSet:0 "SMS_NOTIFICATION"
SADD Saga:activate_prepaid_sim:State:SUCCESS:RequiredTasksSet:0 "DROP_CARD_FROM_INVENTORY"
SET Saga:activate_prepaid_sim:State:COMPENSATED:RequiredTasksSets 1"
```

```
33 SADD Saga:activate_prepaid_sim:State:COMPENSATED:RequiredTasksSet:0 "
       LIBERATE_NUMBER"
34
35 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t1:State {name: 'START'}), (t2:Task {
       name: 'RESERVE_NUMBER'}) CREATE (t1)-[:success]->(t2)"
36 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t1:Task {name: 'RESERVE_NUMBER'}), (
       t2:Task {name: 'CONTRACT_TAKEOVER'}) CREATE (t1)-[:success]->(t2)"
37 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t1:Task {name: 'CONTRACT_TAKEOVER'}),
        (t2:Task {name: 'SIM_CARD_NUMBER_UPDATE'}) CREATE (t1)-[:success]->(t2)"
38 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t1:Task {name: '
       SIM_CARD_NUMBER_UPDATE'}), (t2:Task {name: 'SMS_NOTIFICATION'}) CREATE (t1)-[:
       success]->(t2)"
39 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t1:Task {name: '
       SIM_CARD_NUMBER_UPDATE'}), (t2:Task {name: 'DROP_CARD_FROM_INVENTORY'}) CREATE
       (t1)-[:success]->(t2)"
40 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t1:Task {name: 'SMS_NOTIFICATION'}),
       (t2:State {name: 'SUCCESS'}) CREATE (t1)-[:success]->(t2)"
41 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t1:Task {name: '
       DROP_CARD_FROM_INVENTORY'}), (t2:State {name: 'SUCCESS'}) CREATE (t1)-[:success
       ]->(t2)"
42 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t1:Task {name: 'LIBERATE_NUMBER'}), (
       t2:State {name: 'COMPENSATED'}) CREATE (t1)-[:success]->(t2)"
43
44 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t1:Task {name: 'RESERVE_NUMBER'}), (
       t2:Task {name: 'LIBERATE_NUMBER'}) CREATE (t1)-[:compensated_by]->(t2)"
45
46 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t1:Task {name: 'CONTRACT_TAKEOVER'}),
        (t2:Task {name: 'LIBERATE_NUMBER'}) CREATE (t1)-[:failure]->(t2)"
47
48 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t:Task {name: 'RESERVE_NUMBER'}), (s:
       Service {name: 'NumberManagementService'}) CREATE (t)-[:implemented_by]->(s)"
49 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t:Task {name: 'CONTRACT_TAKEOVER'}),
       (s:Service {name: 'BillingService'}) CREATE (t)-[:implemented_by]->(s)"
50 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t:Task {name: 'SIM_CARD_NUMBER_UPDATE
       '}), (s:Service {name: 'NetworkService'}) CREATE (t)-[:implemented_by]->(s)"
51 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t:Task {name: 'SMS_NOTIFICATION'}), (
       s:Service {name: 'CommunicationsService'}) CREATE (t)-[:implemented_by]->(s)"
52 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t:Task {name: '
       DROP_CARD_FROM_INVENTORY'}), (s:Service {name: 'ERPConnectionService'}) CREATE
       (t)-[:implemented_by]->(s)"
53 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH (t:Task {name: 'LIBERATE_NUMBER'}), (s
       :Service {name: 'NumberManagementService'}) CREATE (t)-[:implemented_by]->(s)"
```

Listing A.3: Example of a saga definition for activating a prepaid SIM card.

Appendix B

Framework's Behaviour Validation

This appendix is to be used as a reference of the Redis commands to use to perform the validations described in section 4.2.1. Observe that in the commands listed in this appendix '<saga_id>' and '<context_id>' stands for the saga identifier and context identifier, respectively. The context identifier is obtained in the response from the Interface upon the saga execution request. Sometimes '<end_state>' is also used and it is meant to be replaced by the end state of the respective validation scenario.

B.1 Commands Definitions

The following is the list of Redis commands to be executed for validating each aspect discussed in the above-mentioned section.

1. Validating whether a context was created:
   1.1. `EXISTS Saga:<saga_id>:<context_id>` - Checks whether the context graph was created.
   1.2. `EXISTS Saga:<saga_id>:<context_id>:state` - Checks whether the context state string was created.
   1.3. `KEYS Saga:<saga_id>:<context_id>:*:CallerTasks` - Lists all entries of the context precedence map.
2. Validating whether all task nodes have been created (verify if the second output is present in the output of the first query):
   2.1. `GRAPH.QUERY Saga:order_food "MATCH p = (s1:State name:'START')-[:success name:'<end_state>') RETURN [n in nodes(p) WHERE n.name <> 'START' AND n.name <> '<end_state>' | n.name]"` - Lists the paths from the start state to the validation scenario's end state in the saga definition graph.
   2.2. `GRAPH.QUERY Saga:<saga_id>:<context_id> "MATCH p = (s1:State name:'START name:'<end_state>') RETURN [n in nodes(p) WHERE n.name <> 'START'`

`AND n.name <> '<end_state>' | n.name]"` - Lists the paths from the start state to the validation scenario's end state in the context graph.

3. Validating the context's final state:

   3.1. `GET Saga:<saga_id>:<context_id>:state` - Returns the final state of the context.

4. Validating the tasks' final states:

   4.1. `GRAPH.QUERY Saga:<saga_id>:<context_id> "MATCH (t:Task) RETURN t.name as taskName, t.technical_sate, t.business_state"` - Lists all tasks in the context graph associated with their technical and business states.

5. Caller Tasks Registrations validation. For every entry of the context precedence map run:

   5.1. `SMEMBERS Saga:<saga_id>:<context_id>:<task_id>:CallerTasks` - Lists the members of an entry of the context precedence map.

   5.2. `KEYS Saga:<saga_id>:Task:<task_id>:RequiredTasksSet:*` - Lists all the required task sets of the saga precedence map entry.

   5.3. `SMEMBERS Saga:<saga_id>:<context_id>:Task:<task_id>:RequiredTasksSet:<i>` - List the members of the sets of the precedence map entry associated with each key of the context precedence map.

B.2 Validation Execution

This section stores the execution outputs of the above Redis commands for every validation scenario presented in 4.2.2. The outputs of the commands will be presented in the same structure as the definitions of the commands.

B.2.1 Order Food Saga (Success)

```
# Context Creation Validation
## Query 1:
EXISTS Saga:order_food:kGRHkT82d0TcV5Hi
## Output 1:
(integer) 1

## Query 2:
EXISTS Saga:order_food:kGRHkT82d0TcV5Hi:state
## Output 2:
(integer) 1

## Query 3:
```

```
KEYS Saga:order_food:kGRHkT82d0TcV5Hi:*:CallerTasks
## Output 3:
1) "Saga:order_food:kGRHkT82d0TcV5Hi:CREATE_TICKET:CallerTasks"
2) "Saga:order_food:kGRHkT82d0TcV5Hi:CREATE_ORDER:CallerTasks"
3) "Saga:order_food:kGRHkT82d0TcV5Hi:APPROVE_ORDER:CallerTasks"
4) "Saga:order_food:kGRHkT82d0TcV5Hi:APPROVE_TICKET:CallerTasks"
5) "Saga:order_food:kGRHkT82d0TcV5Hi:State:SUCCESS:CallerTasks"
6) "Saga:order_food:kGRHkT82d0TcV5Hi:AUTHORIZE_CARD:CallerTasks"
7) "Saga:order_food:kGRHkT82d0TcV5Hi:VERIFY_CUSTOMER:CallerTasks"

# Tasks Creation Validation
## Query 4:
GRAPH.QUERY Saga:order_food "MATCH p = (s1:State {name:'START'})-[:success|:failure
    *]->(S2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.name <> 'START'
    AND n.name <> 'SUCCESS' | n.name]"
## Output 4:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
2) 1) 1) "[CREATE_ORDER, VERIFY_CUSTOMER, CREATE_TICKET, AUTHORIZE_CARD,
    APPROVE_TICKET, APPROVE_ORDER]"
3) 1) "Cached execution: 1"
   2) "Query internal execution time: 0.492664 milliseconds"

## Query 5:
GRAPH.QUERY Saga:order_food:kGRHkT82d0TcV5Hi "MATCH p = (s1:State {name:'START'})
    -[*]->(s2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.name <> 'START'
     AND n.name <> 'SUCCESS' | n.name]"
## Output 5:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
2) 1) 1) "[CREATE_ORDER, VERIFY_CUSTOMER, CREATE_TICKET, AUTHORIZE_CARD,
    APPROVE_TICKET, APPROVE_ORDER]"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 1.040709 milliseconds"

### Note: Output 5 is present in Output 4.

# Context Final State Validation
## Query 6:
GET Saga:order_food:kGRHkT82d0TcV5Hi:state
## Output 6:
"SUCCESS"

# Tasks Final State Validation
## Query 7:
GRAPH.QUERY Saga:order_food:kGRHkT82d0TcV5Hi "MATCH (t:Task) RETURN t.name as
    taskName, t.technical_state, t.business_state"
## Output 7:
1) 1) "taskName"
   2) "t.technical_state"
   3) "t.business_state"
```

```
2) 1) 1) "CREATE_ORDER"
      2) "ROUTED"
      3) "SUCCESS"
   2) 1) "VERIFY_CUSTOMER"
      2) "ROUTED"
      3) "SUCCESS"
   3) 1) "CREATE_TICKET"
      2) "ROUTED"
      3) "SUCCESS"
   4) 1) "AUTHORIZE_CARD"
      2) "ROUTED"
      3) "SUCCESS"
   5) 1) "APPROVE_TICKET"
      2) "ROUTED"
      3) "SUCCESS"
   6) 1) "APPROVE_ORDER"
      2) "ROUTED"
      3) "SUCCESS"
3) 1) "Cached execution: 1"
   2) "Query internal execution time: 0.359180 milliseconds"

# Caller Tasks Registrations Validation
### Note: The keys of the context precedence map are available in output 3.
## Queries 8:
SMEMBERS Saga:order_food:kGRHkT82d0TcV5Hi:CREATE_TICKET:CallerTasks
SMEMBERS Saga:order_food:kGRHkT82d0TcV5Hi:CREATE_ORDER:CallerTasks
SMEMBERS Saga:order_food:kGRHkT82d0TcV5Hi:APPROVE_ORDER:CallerTasks
SMEMBERS Saga:order_food:kGRHkT82d0TcV5Hi:APPROVE_TICKET:CallerTasks
SMEMBERS Saga:order_food:kGRHkT82d0TcV5Hi:State:SUCCESS:CallerTasks
SMEMBERS Saga:order_food:kGRHkT82d0TcV5Hi:AUTHORIZE_CARD:CallerTasks
SMEMBERS Saga:order_food:kGRHkT82d0TcV5Hi:VERIFY_CUSTOMER:CallerTasks
## Outputs 8 (in the same order as the queries):
1) "VERIFY_CUSTOMER"
1) "START"
1) "APPROVE_TICKET"
1) "AUTHORIZE_CARD"
1) "APPROVE_ORDER"
1) "CREATE_TICKET"
1) "CREATE_ORDER"

## Queries 9:
KEYS Saga:order_food:Task:CREATE_TICKET:RequiredTasksSet:*
KEYS Saga:order_food:Task:CREATE_ORDER:RequiredTasksSet:*
KEYS Saga:order_food:Task:APPROVE_ORDER:RequiredTasksSet:*
KEYS Saga:order_food:Task:APPROVE_TICKET:RequiredTasksSet:*
KEYS Saga:order_food:State:SUCCESS:RequiredTasksSet:*
KEYS Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSet:*
KEYS Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSet:*
## Outputs 9:
```

```
1) "Saga:order_food:Task:CREATE_TICKET:RequiredTasksSet:0"
1) "Saga:order_food:Task:CREATE_ORDER:RequiredTasksSet:0"
1) "Saga:order_food:Task:APPROVE_ORDER:RequiredTasksSet:0"
1) "Saga:order_food:Task:APPROVE_TICKET:RequiredTasksSet:0"
1) "Saga:order_food:State:SUCCESS:RequiredTasksSet:0"
1) "Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSet:0"
1) "Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSet:0"

## Queries 10:
SMEMBERS Saga:order_food:Task:CREATE_TICKET:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:CREATE_ORDER:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:APPROVE_ORDER:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:APPROVE_TICKET:RequiredTasksSet:0
SMEMBERS Saga:order_food:State:SUCCESS:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSet:0
## Outputs 10:
1) "VERIFY_CUSTOMER"
1) "START"
1) "APPROVE_TICKET"
1) "AUTHORIZE_CARD"
1) "APPROVE_ORDER"
1) "CREATE_TICKET"
1) "CREATE_ORDER"

### Note: Outputs 10 and 8 are equal.
```

Listing B.1: PoC's behaviour validation for the successful execution of the order food saga.

B.2.2 Order Food Saga (Failure)

```
# Context Creation Validation
## Query 1:
EXISTS Saga:order_food:jHImx53tPTmqYzjH
## Output 1:
(integer) 1

## Query 2:
EXISTS Saga:order_food:jHImx53tPTmqYzjH:state
## Output 2:
(integer) 1

## Query 3:
KEYS Saga:order_food:jHImx53tPTmqYzjH:*:CallerTasks
## Output 3:
1) "Saga:order_food:jHImx53tPTmqYzjH:CREATE_TICKET:CallerTasks"
2) "Saga:order_food:jHImx53tPTmqYzjH:CREATE_ORDER:CallerTasks"
```

```
3) "Saga:order_food:jHImx53tPTmqYzjH:REJECT_ORDER:CallerTasks"
4) "Saga:order_food:jHImx53tPTmqYzjH:REJECT_TICKET:CallerTasks"
5) "Saga:order_food:jHImx53tPTmqYzjH:State:COMPENSATED:CallerTasks"
6) "Saga:order_food:jHImx53tPTmqYzjH:AUTHORIZE_CARD:CallerTasks"
7) "Saga:order_food:jHImx53tPTmqYzjH:VERIFY_CUSTOMER:CallerTasks"

# Tasks Creation Validation
## Query 4:
GRAPH.QUERY Saga:order_food "MATCH p = (s1:State {name:'START'})-[:success|:failure
    *]->(S2:State {name:'COMPENSATED'}) RETURN [n in nodes(p) WHERE n.name <> '
    START' AND n.name <> 'COMPENSATED' | n.name]"
## Output 4:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'COMPENSATED' | n.name]
    "
2) 1) 1) "[CREATE_ORDER, VERIFY_CUSTOMER, CREATE_TICKET, AUTHORIZE_CARD,
    REJECT_TICKET, REJECT_ORDER]"
3) 1) "Cached execution: 1"
   2) "Query internal execution time: 0.492664 milliseconds"

## Query 5:
GRAPH.QUERY Saga:order_food:jHImx53tPTmqYzjH "MATCH p = (s1:State {name:'START'})
    -[*]->(s2:State {name:'COMPENSATED'}) RETURN [n in nodes(p) WHERE n.name <> '
    START' AND n.name <> 'COMPENSATED' | n.name]"
## Output 5:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'COMPENSATED' | n.name]
    "
2) 1) 1) "[CREATE_ORDER, VERIFY_CUSTOMER, REJECT_ORDER]"
   2) 1) "[CREATE_ORDER, VERIFY_CUSTOMER, CREATE_TICKET, AUTHORIZE_CARD,
       REJECT_TICKET, REJECT_ORDER]"
3) 1) "Cached execution: 1"
   2) "Query internal execution time: 0.390250 milliseconds"

### Note: Output 5 is present in Output 4.

# Context Final State Validation
## Query 6:
GET Saga:order_food:jHImx53tPTmqYzjH:state
## Output 6:
"COMPENSATED"

# Tasks Final State Validation
## Query 7:
GRAPH.QUERY Saga:order_food:jHImx53tPTmqYzjH "MATCH (t:Task) RETURN t.name as
    taskName, t.technical_state, t.business_state"
## Output 7:
1) 1) "taskName"
   2) "t.technical_state"
   3) "t.business_state"
2) 1) 1) "CREATE_ORDER"
```

```
      2) "ROUTED"
      3) "COMPENSATED"
   2) 1) "VERIFY_CUSTOMER"
      2) "ROUTED"
      3) "SUCCESS"
   3) 1) "CREATE_TICKET"
      2) "ROUTED"
      3) "COMPENSATED"
   4) 1) "AUTHORIZE_CARD"
      2) "ROUTED"
      3) "FAILED"
   5) 1) "REJECT_TICKET"
      2) "ROUTED"
      3) "SUCCESS"
   6) 1) "REJECT_ORDER"
      2) "ROUTED"
      3) "SUCCESS"
3) 1) "Cached execution: 1"
   2) "Query internal execution time: 0.359180 milliseconds"

# Caller Tasks Registrations Validation
### Note: The keys of the context precedence map are available in output 3.
## Queries 8:
SMEMBERS Saga:order_food:jHImx53tPTmqYzjH:CREATE_TICKET:CallerTasks
SMEMBERS Saga:order_food:jHImx53tPTmqYzjH:CREATE_ORDER:CallerTasks
SMEMBERS Saga:order_food:jHImx53tPTmqYzjH:REJECT_ORDER:CallerTasks
SMEMBERS Saga:order_food:jHImx53tPTmqYzjH:REJECT_TICKET:CallerTasks
SMEMBERS Saga:order_food:jHImx53tPTmqYzjH:State:COMPENSATED:CallerTasks
SMEMBERS Saga:order_food:jHImx53tPTmqYzjH:AUTHORIZE_CARD:CallerTasks
SMEMBERS Saga:order_food:jHImx53tPTmqYzjH:VERIFY_CUSTOMER:CallerTasks
## Outputs 8 (in the same order as the queries):
1) "VERIFY_CUSTOMER"
1) "START"
1) "REJECT_TICKET"
1) "AUTHORIZE_CARD"
1) "REJECT_ORDER"
1) "CREATE_TICKET"
1) "CREATE_ORDER"

## Queries 9:
KEYS Saga:order_food:Task:CREATE_TICKET:RequiredTasksSet:*
KEYS Saga:order_food:Task:CREATE_ORDER:RequiredTasksSet:*
KEYS Saga:order_food:Task:REJECT_ORDER:RequiredTasksSet:*
KEYS Saga:order_food:Task:REJECT_TICKET:RequiredTasksSet:*
KEYS Saga:order_food:State:COMPENSATED:RequiredTasksSet:*
KEYS Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSet:*
KEYS Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSet:*
## Outputs 9:
1) "Saga:order_food:Task:CREATE_TICKET:RequiredTasksSet:0"
```

```
1) "Saga:order_food:Task:CREATE_ORDER:RequiredTasksSet:0"
1) "Saga:order_food:Task:REJECT_ORDER:RequiredTasksSet:0"
2) "Saga:order_food:Task:REJECT_ORDER:RequiredTasksSet:1"
1) "Saga:order_food:Task:REJECT_TICKET:RequiredTasksSet:0"
1) "Saga:order_food:State:COMPENSATED:RequiredTasksSet:0"
1) "Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSet:0"
1) "Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSet:0"

## Queries 10:
SMEMBERS Saga:order_food:Task:CREATE_TICKET:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:CREATE_ORDER:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:REJECT_ORDER:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:REJECT_ORDER:RequiredTasksSet:1
SMEMBERS Saga:order_food:Task:REJECT_TICKET:RequiredTasksSet:0
SMEMBERS Saga:order_food:State:COMPENSATED:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSet:0
## Outputs 10:
1) "VERIFY_CUSTOMER"
1) "START"
1) "VERIFY_CUSTOMER"
1) "REJECT_TICKET"
1) "AUTHORIZE_CARD"
1) "REJECT_ORDER"
1) "CREATE_TICKET"
1) "CREATE_ORDER"

### Note: The entries of Output 8 are contained in Output 10.
```

Listing B.2: PoC's behaviour validation for the unsuccessful execution of the order food saga.

B.2.3 Parallel Order Food Saga (Success)

```
# Context Creation Validation
## Query 1:
EXISTS Saga:order_food:3gJZkap3u9FFp8um
## Output 1:
(integer) 1

## Query 2:
EXISTS Saga:order_food:3gJZkap3u9FFp8um:state
## Output 2:
(integer) 1

## Query 3:
KEYS Saga:order_food:3gJZkap3u9FFp8um:*:CallerTasks
## Output 3:
```

```
1) "Saga:order_food:3gJZkap3u9FFp8um:AUTHORIZE_CARD:CallerTasks"
2) "Saga:order_food:3gJZkap3u9FFp8um:CREATE_TICKET:CallerTasks"
3) "Saga:order_food:3gJZkap3u9FFp8um:APPROVE_TICKET:CallerTasks"
4) "Saga:order_food:3gJZkap3u9FFp8um:VERIFY_CUSTOMER:CallerTasks"
5) "Saga:order_food:3gJZkap3u9FFp8um:CREATE_ORDER:CallerTasks"
6) "Saga:order_food:3gJZkap3u9FFp8um:State:SUCCESS:CallerTasks"
7) "Saga:order_food:3gJZkap3u9FFp8um:APPROVE_ORDER:CallerTasks"

# Tasks Creation Validation
## Query 4:
GRAPH.QUERY Saga:order_food "MATCH p = (s1:State {name:'START'})-[:success|:failure
    *]->(S2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.name <> 'START'
    AND n.name <> 'SUCCESS' | n.name]"
## Output 4:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
2) 1) 1) "[CREATE_TICKET, AUTHORIZE_CARD, APPROVE_ORDER]"
   2) 1) "[CREATE_TICKET, AUTHORIZE_CARD, APPROVE_TICKET]"
   3) 1) "[VERIFY_CUSTOMER, AUTHORIZE_CARD, APPROVE_ORDER]"
   4) 1) "[VERIFY_CUSTOMER, AUTHORIZE_CARD, APPROVE_TICKET]"
   5) 1) "[CREATE_ORDER, AUTHORIZE_CARD, APPROVE_ORDER]"
   6) 1) "[CREATE_ORDER, AUTHORIZE_CARD, APPROVE_TICKET]"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 0.924294 milliseconds"

## Query 5:
GRAPH.QUERY Saga:order_food:3gJZkap3u9FFp8um "MATCH p = (s1:State {name:'START'})
    -[*]->(s2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.name <> 'START'
     AND n.name <> 'SUCCESS' | n.name]"
## Output 5:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
2) 1) 1) "[CREATE_TICKET, AUTHORIZE_CARD, APPROVE_ORDER]"
   2) 1) "[CREATE_TICKET, AUTHORIZE_CARD, APPROVE_TICKET]"
   3) 1) "[VERIFY_CUSTOMER, AUTHORIZE_CARD, APPROVE_ORDER]"
   4) 1) "[VERIFY_CUSTOMER, AUTHORIZE_CARD, APPROVE_TICKET]"
   5) 1) "[CREATE_ORDER, AUTHORIZE_CARD, APPROVE_ORDER]"
   6) 1) "[CREATE_ORDER, AUTHORIZE_CARD, APPROVE_TICKET]"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 0.767681 milliseconds"

### Note: Output 5 is present in Output 4.

# Context Final State Validation
## Query 6:
GET Saga:order_food:3gJZkap3u9FFp8um:state
## Output 6:
"SUCCESS"

# Tasks Final State Validation
## Query 7:
```

```
GRAPH.QUERY Saga:order_food:3gJZkap3u9FFp8um "MATCH (t:Task) RETURN t.name as
    taskName, t.technical_state, t.business_state"
## Output 7:
1) 1) "taskName"
   2) "t.technical_state"
   3) "t.business_state"
2) 1) 1) "CREATE_ORDER"
      2) "ROUTED"
      3) "SUCCESS"
   2) 1) "VERIFY_CUSTOMER"
      2) "ROUTED"
      3) "SUCCESS"
   3) 1) "CREATE_TICKET"
      2) "ROUTED"
      3) "SUCCESS"
   4) 1) "AUTHORIZE_CARD"
      2) "ROUTED"
      3) "SUCCESS"
   5) 1) "APPROVE_TICKET"
      2) "ROUTED"
      3) "SUCCESS"
   6) 1) "APPROVE_ORDER"
      2) "ROUTED"
      3) "SUCCESS"
3) 1) "Cached execution: 1"
   2) "Query internal execution time: 0.359180 milliseconds"

# Caller Tasks Registrations Validation
### Note: The keys of the context precedence map are available in output 3.
## Queries 8:
SMEMBERS Saga:order_food:3gJZkap3u9FFp8um:CREATE_TICKET:CallerTasks
SMEMBERS Saga:order_food:3gJZkap3u9FFp8um:CREATE_ORDER:CallerTasks
SMEMBERS Saga:order_food:3gJZkap3u9FFp8um:APPROVE_ORDER:CallerTasks
SMEMBERS Saga:order_food:3gJZkap3u9FFp8um:APPROVE_TICKET:CallerTasks
SMEMBERS Saga:order_food:3gJZkap3u9FFp8um:State:SUCCESS:CallerTasks
SMEMBERS Saga:order_food:3gJZkap3u9FFp8um:AUTHORIZE_CARD:CallerTasks
SMEMBERS Saga:order_food:3gJZkap3u9FFp8um:VERIFY_CUSTOMER:CallerTasks
## Outputs 8 (in the same order as the queries):
1) "START"
1) "START"
1) "AUTHORIZE_CARD"
1) "AUTHORIZE_CARD"
1) "APPROVE_TICKET"
2) "APPROVE_ORDER"
1) "CREATE_TICKET"
2) "VERIFY_CUSTOMER"
3) "CREATE_ORDER"
1) "START"

```

```
## Queries 9:
KEYS Saga:order_food:Task:CREATE_TICKET:RequiredTasksSet:*
KEYS Saga:order_food:Task:CREATE_ORDER:RequiredTasksSet:*
KEYS Saga:order_food:Task:APPROVE_ORDER:RequiredTasksSet:*
KEYS Saga:order_food:Task:APPROVE_TICKET:RequiredTasksSet:*
KEYS Saga:order_food:State:SUCCESS:RequiredTasksSet:*
KEYS Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSet:*
KEYS Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSet:*
## Outputs 9:
1) "Saga:order_food:Task:CREATE_TICKET:RequiredTasksSet:0"
1) "Saga:order_food:Task:CREATE_ORDER:RequiredTasksSet:0"
1) "Saga:order_food:Task:APPROVE_ORDER:RequiredTasksSet:0"
1) "Saga:order_food:Task:APPROVE_TICKET:RequiredTasksSet:0"
1) "Saga:order_food:State:SUCCESS:RequiredTasksSet:0"
1) "Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSet:0"
1) "Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSet:0"

## Queries 10:
SMEMBERS Saga:order_food:Task:CREATE_TICKET:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:CREATE_ORDER:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:APPROVE_ORDER:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:APPROVE_TICKET:RequiredTasksSet:0
SMEMBERS Saga:order_food:State:SUCCESS:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSet:0
## Outputs 10:
1) "START"
1) "START"
1) "AUTHORIZE_CARD"
1) "AUTHORIZE_CARD"
1) "APPROVE_TICKET"
2) "APPROVE_ORDER"
1) "CREATE_TICKET"
2) "VERIFY_CUSTOMER"
3) "CREATE_ORDER"
1) "START"

### Note: Outputs 10 and 8 are equal.
```

Listing B.3: PoC's behaviour validation for the successful execution of the parallel order food saga.

B.2.4 Parallel Order Food Saga (Failure)

```
# Context Creation Validation
## Query 1:
EXISTS Saga:order_food:M1HFK0NstryflsMY
```

```
## Output 1:
(integer) 1

## Query 2:
EXISTS Saga:order_food:M1HFK0NstryflsMY:state
## Output 2:
(integer) 1

## Query 3:
KEYS Saga:order_food:M1HFK0NstryflsMY:*:CallerTasks
## Output 3:
1) "Saga:order_food:M1HFK0NstryflsMY:REJECT_TICKET:CallerTasks"
2) "Saga:order_food:M1HFK0NstryflsMY:REJECT_ORDER:CallerTasks"
3) "Saga:order_food:M1HFK0NstryflsMY:AUTHORIZE_CARD:CallerTasks"
4) "Saga:order_food:M1HFK0NstryflsMY:CREATE_ORDER:CallerTasks"
5) "Saga:order_food:M1HFK0NstryflsMY:VERIFY_CUSTOMER:CallerTasks"
6) "Saga:order_food:M1HFK0NstryflsMY:State:COMPENSATED:CallerTasks"
7) "Saga:order_food:M1HFK0NstryflsMY:CREATE_TICKET:CallerTasks"

# Tasks Creation Validation
## Query 4:
GRAPH.QUERY Saga:order_food "MATCH p = (s1:State {name:'START'})-[:success|:failure
    *]->(S2:State {name:'COMPENSATED'}) RETURN [n in nodes(p) WHERE n.name <> '
    START' AND n.name <> 'COMPENSATED' | n.name]"
## Output 4:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'COMPENSATED' | n.name]
    "
2) 1) 1) "[CREATE_TICKET, AUTHORIZE_CARD, REJECT_TICKET]"
   2) 1) "[CREATE_TICKET, AUTHORIZE_CARD, REJECT_ORDER]"
   3) 1) "[VERIFY_CUSTOMER, AUTHORIZE_CARD, REJECT_TICKET]"
   4) 1) "[VERIFY_CUSTOMER, AUTHORIZE_CARD, REJECT_ORDER]"
   5) 1) "[CREATE_ORDER, AUTHORIZE_CARD, REJECT_TICKET]"
   6) 1) "[CREATE_ORDER, AUTHORIZE_CARD, REJECT_ORDER]"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 1.241604 milliseconds"

## Query 5:
GRAPH.QUERY Saga:order_food:M1HFK0NstryflsMY "MATCH p = (s1:State {name:'START'})
    -[*]->(s2:State {name:'COMPENSATED'}) RETURN [n in nodes(p) WHERE n.name <> '
    START' AND n.name <> 'COMPENSATED' | n.name]"
## Output 5:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'COMPENSATED' | n.name]
    "
2) 1) 1) "[CREATE_TICKET, AUTHORIZE_CARD, REJECT_TICKET]"
   2) 1) "[CREATE_TICKET, AUTHORIZE_CARD, REJECT_ORDER]"
   3) 1) "[VERIFY_CUSTOMER, AUTHORIZE_CARD, REJECT_TICKET]"
   4) 1) "[VERIFY_CUSTOMER, AUTHORIZE_CARD, REJECT_ORDER]"
   5) 1) "[CREATE_ORDER, AUTHORIZE_CARD, REJECT_TICKET]"
   6) 1) "[CREATE_ORDER, AUTHORIZE_CARD, REJECT_ORDER]"
```

```
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 0.711702 milliseconds"

### Note: Output 5 is present in Output 4.

# Context Final State Validation
## Query 6:
GET Saga:order_food:M1HFK0NstryflsMY:state
## Output 6:
"COMPENSATED"

# Tasks Final State Validation
## Query 7:
GRAPH.QUERY Saga:order_food:M1HFK0NstryflsMY "MATCH (t:Task) RETURN t.name as
    taskName, t.technical_state, t.business_state"
## Output 7:
1) 1) "taskName"
   2) "t.technical_state"
   3) "t.business_state"
2) 1) 1) "CREATE_ORDER"
      2) "ROUTED"
      3) "COMPENSATED"
   2) 1) "VERIFY_CUSTOMER"
      2) "ROUTED"
      3) "COMPENSATED"
   3) 1) "CREATE_TICKET"
      2) "ROUTED"
      3) "COMPENSATED"
   4) 1) "AUTHORIZE_CARD"
      2) "ROUTED"
      3) "FAILED"
   5) 1) "REJECT_ORDER"
      2) "ROUTED"
      3) "SUCCESS"
   6) 1) "REJECT_TICKET"
      2) "ROUTED"
      3) "SUCCESS"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 0.335295 milliseconds"

# Caller Tasks Registrations Validation
### Note: The keys of the context precedence map are available in output 3.
## Queries 8:
SMEMBERS Saga:order_food:M1HFK0NstryflsMY:CREATE_TICKET:CallerTasks
SMEMBERS Saga:order_food:M1HFK0NstryflsMY:CREATE_ORDER:CallerTasks
SMEMBERS Saga:order_food:M1HFK0NstryflsMY:REJECT_ORDER:CallerTasks
SMEMBERS Saga:order_food:M1HFK0NstryflsMY:REJECT_TICKET:CallerTasks
SMEMBERS Saga:order_food:M1HFK0NstryflsMY:State:COMPENSATED:CallerTasks
SMEMBERS Saga:order_food:M1HFK0NstryflsMY:AUTHORIZE_CARD:CallerTasks
```

```
SMEMBERS Saga:order_food:M1HFK0NstryflsMY:VERIFY_CUSTOMER:CallerTasks
## Outputs 8 (in the same order as the queries):
1) "START"
1) "START"
1) "AUTHORIZE_CARD"
1) "AUTHORIZE_CARD"
1) "REJECT_ORDER"
2) "REJECT_TICKET"
1) "CREATE_TICKET"
2) "VERIFY_CUSTOMER"
3) "CREATE_ORDER"
1) "START"

## Queries 9:
KEYS Saga:order_food:Task:CREATE_TICKET:RequiredTasksSet:*
KEYS Saga:order_food:Task:CREATE_ORDER:RequiredTasksSet:*
KEYS Saga:order_food:Task:REJECT_ORDER:RequiredTasksSet:*
KEYS Saga:order_food:Task:REJECT_TICKET:RequiredTasksSet:*
KEYS Saga:order_food:State:COMPENSATED:RequiredTasksSet:*
KEYS Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSet:*
KEYS Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSet:*
## Outputs 9:
1) "Saga:order_food:Task:CREATE_TICKET:RequiredTasksSet:0"
1) "Saga:order_food:Task:CREATE_ORDER:RequiredTasksSet:0"
1) "Saga:order_food:Task:REJECT_ORDER:RequiredTasksSet:0"
1) "Saga:order_food:Task:REJECT_TICKET:RequiredTasksSet:0"
1) "Saga:order_food:State:COMPENSATED:RequiredTasksSet:0"
1) "Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSet:0"
1) "Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSet:0"

## Queries 10:
SMEMBERS Saga:order_food:Task:CREATE_TICKET:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:CREATE_ORDER:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:REJECT_ORDER:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:REJECT_ORDER:RequiredTasksSet:1
SMEMBERS Saga:order_food:Task:REJECT_TICKET:RequiredTasksSet:0
SMEMBERS Saga:order_food:State:COMPENSATED:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:AUTHORIZE_CARD:RequiredTasksSet:0
SMEMBERS Saga:order_food:Task:VERIFY_CUSTOMER:RequiredTasksSet:0
## Outputs 10:
1) "START"
1) "START"
1) "AUTHORIZE_CARD"
1) "AUTHORIZE_CARD"
1) "REJECT_ORDER"
2) "REJECT_TICKET"
1) "CREATE_TICKET"
2) "VERIFY_CUSTOMER"
3) "CREATE_ORDER"
```

```
144 1) "START"
145
146 ### Note: The entries of Output 8 are contained in Output 10.
```

Listing B.4: PoC's behaviour validation for the unsuccessful execution of the parallel order food saga.

B.2.5 Activate Prepaid SIM Card Saga (Success)

```
1  # Context Creation Validation
2  ## Query 1:
3  EXISTS Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ
4  ## Output 1:
5  (integer) 1
6
7  ## Query 2:
8  EXISTS Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:state
9  ## Output 2:
10 (integer) 1
11
12 ## Query 3:
13 KEYS Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:*:CallerTasks
14 ## Output 3:
15 1) "Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:State:SUCCESS:CallerTasks"
16 2) "Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:RESERVE_NUMBER:CallerTasks"
17 3) "Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:DROP_CARD_FROM_INVENTORY:CallerTasks
       "
18 4) "Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:CONTRACT_TAKEOVER:CallerTasks"
19 5) "Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:SIM_CARD_NUMBER_UPDATE:CallerTasks"
20 6) "Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:SMS_NOTIFICATION:CallerTasks"
21
22 # Tasks Creation Validation
23 ## Query 4:
24 GRAPH.QUERY Saga:activate_prepaid_sim "MATCH p = (s1:State {name:'START'})-[:
       success|:failure*]->(S2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.
       name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
25 ## Output 4:
26 1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
27 2) 1) 1) "[RESERVE_NUMBER, CONTRACT_TAKEOVER, SIM_CARD_NUMBER_UPDATE,
       DROP_CARD_FROM_INVENTORY]"
28    2) 1) "[RESERVE_NUMBER, CONTRACT_TAKEOVER, SIM_CARD_NUMBER_UPDATE,
          SMS_NOTIFICATION]"
29 3) 1) "Cached execution: 0"
30    2) "Query internal execution time: 1.114889 milliseconds"
31
32 ## Query 5:
```

```
GRAPH.QUERY Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ "MATCH p = (s1:State {name:'START'})-[*]->(s2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
## Output 5:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
2) 1) 1) "[RESERVE_NUMBER, CONTRACT_TAKEOVER, SIM_CARD_NUMBER_UPDATE, DROP_CARD_FROM_INVENTORY]"
   2) 1) "[RESERVE_NUMBER, CONTRACT_TAKEOVER, SIM_CARD_NUMBER_UPDATE, SMS_NOTIFICATION]"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 1.063129 milliseconds"

### Note: Output 5 is present in Output 4.

# Context Final State Validation
## Query 6:
GET Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:state
## Output 6:
"SUCCESS"

# Tasks Final State Validation
## Query 7:
GRAPH.QUERY Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ "MATCH (t:Task) RETURN t.name as taskName, t.technical_state, t.business_state"
## Output 7:
1) 1) "taskName"
   2) "t.technical_state"
   3) "t.business_state"
2) 1) 1) "RESERVE_NUMBER"
      2) "ROUTED"
      3) "SUCCESS"
   2) 1) "CONTRACT_TAKEOVER"
      2) "ROUTED"
      3) "SUCCESS"
   3) 1) "SIM_CARD_NUMBER_UPDATE"
      2) "ROUTED"
      3) "SUCCESS"
   4) 1) "SMS_NOTIFICATION"
      2) "ROUTED"
      3) "SUCCESS"
   5) 1) "DROP_CARD_FROM_INVENTORY"
      2) "ROUTED"
      3) "SUCCESS"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 0.325233 milliseconds"

# Caller Tasks Registrations Validation
### Note: The keys of the context precedence map are available in output 3.
## Queries 8:
```

```
77  SMEMBERS Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:State:SUCCESS:CallerTasks
78  SMEMBERS Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:RESERVE_NUMBER:CallerTasks
79  SMEMBERS Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:DROP_CARD_FROM_INVENTORY:
        CallerTasks
80  SMEMBERS Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:CONTRACT_TAKEOVER:CallerTasks
81  SMEMBERS Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:SIM_CARD_NUMBER_UPDATE:
        CallerTasks
82  SMEMBERS Saga:activate_prepaid_sim:l9LoARPiUmsfz4zQ:SMS_NOTIFICATION:CallerTasks
83  ## Outputs 8 (in the same order as the queries):
84  1) "DROP_CARD_FROM_INVENTORY"
85  2) "SMS_NOTIFICATION"
86  1) "START"
87  1) "SIM_CARD_NUMBER_UPDATE"
88  1) "RESERVE_NUMBER"
89  1) "CONTRACT_TAKEOVER"
90  1) "SIM_CARD_NUMBER_UPDATE"
91
92  ## Queries 9:
93  KEYS Saga:activate_prepaid_sim:State:SUCCESS:RequiredTasksSet:*
94  KEYS Saga:activate_prepaid_sim:Task:RESERVE_NUMBER:RequiredTasksSet:*
95  KEYS Saga:activate_prepaid_sim:Task:DROP_CARD_FROM_INVENTORY:RequiredTasksSet:*
96  KEYS Saga:activate_prepaid_sim:Task:CONTRACT_TAKEOVER:RequiredTasksSet:*
97  KEYS Saga:activate_prepaid_sim:Task:SIM_CARD_NUMBER_UPDATE:RequiredTasksSet:*
98  KEYS Saga:activate_prepaid_sim:Task:SMS_NOTIFICATION:RequiredTasksSet:*
99  ## Outputs 9:
100 1) "Saga:activate_prepaid_sim:State:SUCCESS:RequiredTasksSet:0"
101 1) "Saga:activate_prepaid_sim:Task:RESERVE_NUMBER:RequiredTasksSet:0"
102 1) "Saga:activate_prepaid_sim:Task:DROP_CARD_FROM_INVENTORY:RequiredTasksSet:0"
103 1) "Saga:activate_prepaid_sim:Task:CONTRACT_TAKEOVER:RequiredTasksSet:0"
104 1) "Saga:activate_prepaid_sim:Task:SIM_CARD_NUMBER_UPDATE:RequiredTasksSet:0"
105 1) "Saga:activate_prepaid_sim:Task:SMS_NOTIFICATION:RequiredTasksSet:0"
106
107 ## Queries 10:
108 SMEMBERS Saga:activate_prepaid_sim:State:SUCCESS:RequiredTasksSet:0
109 SMEMBERS Saga:activate_prepaid_sim:Task:RESERVE_NUMBER:RequiredTasksSet:0
110 SMEMBERS Saga:activate_prepaid_sim:Task:DROP_CARD_FROM_INVENTORY:RequiredTasksSet:0
111 SMEMBERS Saga:activate_prepaid_sim:Task:CONTRACT_TAKEOVER:RequiredTasksSet:0
112 SMEMBERS Saga:activate_prepaid_sim:Task:SIM_CARD_NUMBER_UPDATE:RequiredTasksSet:0
113 SMEMBERS Saga:activate_prepaid_sim:Task:SMS_NOTIFICATION:RequiredTasksSet:0
114 ## Outputs 10:
115 1) "DROP_CARD_FROM_INVENTORY"
116 2) "SMS_NOTIFICATION"
117 1) "START"
118 1) "SIM_CARD_NUMBER_UPDATE"
119 1) "RESERVE_NUMBER"
120 1) "CONTRACT_TAKEOVER"
121 1) "SIM_CARD_NUMBER_UPDATE"
122
123 ### Note: Outputs 10 and 8 are equal.
```

Listing B.5: PoC's behaviour validation for the successful execution of the activate prepaid SIM card saga.

B.2.6 Activate Prepaid SIM Card Saga (Failure)

```
# Context Creation Validation
## Query 1:
EXISTS Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO
## Output 1:
(integer) 1

## Query 2:
EXISTS Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO:state
## Output 2:
(integer) 1

## Query 3:
KEYS Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO:*:CallerTasks
## Output 3:
1) "Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO:RESERVE_NUMBER:CallerTasks"
2) "Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO:LIBERATE_NUMBER:CallerTasks"
3) "Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO:State:COMPENSATED:CallerTasks"
4) "Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO:CONTRACT_TAKEOVER:CallerTasks"

# Tasks Creation Validation
## Query 4:
GRAPH.QUERY Saga:activate_prepaid_sim "MATCH p = (s1:State {name:'START'})-[:
    success|:failure*]->(S2:State {name:'COMPENSATED'}) RETURN [n in nodes(p) WHERE
     n.name <> 'START' AND n.name <> 'COMPENSATED' | n.name]"
## Output 4:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'COMPENSATED' | n.name]
    "
2) 1) 1) "[RESERVE_NUMBER, CONTRACT_TAKEOVER, LIBERATE_NUMBER]"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 0.782847 milliseconds"

## Query 5:
GRAPH.QUERY Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO "MATCH p = (s1:State {name:'
    START'})-[*]->(s2:State {name:'COMPENSATED'}) RETURN [n in nodes(p) WHERE n.
    name <> 'START' AND n.name <> 'COMPENSATED' | n.name]"
## Output 5:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'COMPENSATED' | n.name]
    "
2) 1) 1) "[RESERVE_NUMBER, CONTRACT_TAKEOVER, LIBERATE_NUMBER]"
3) 1) "Cached execution: 0"
```

```
   2) "Query internal execution time: 0.812957 milliseconds"

### Note: Output 5 is present in Output 4.

# Context Final State Validation
## Query 6:
GET Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO:state
## Output 6:
"COMPENSATED"

# Tasks Final State Validation
## Query 7:
GRAPH.QUERY Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO "MATCH (t:Task) RETURN t.
    name as taskName, t.technical_state, t.business_state"
## Output 7:
1) 1) "taskName"
   2) "t.technical_state"
   3) "t.business_state"
2) 1) 1) "RESERVE_NUMBER"
      2) "ROUTED"
      3) "COMPENSATED"
   2) 1) "CONTRACT_TAKEOVER"
      2) "ROUTED"
      3) "FAILED"
   3) 1) "LIBERATE_NUMBER"
      2) "ROUTED"
      3) "SUCCESS"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 0.285487 milliseconds"

# Caller Tasks Registrations Validation
### Note: The keys of the context precedence map are available in output 3.
## Queries 8:
SMEMBERS Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO:RESERVE_NUMBER:CallerTasks
SMEMBERS Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO:LIBERATE_NUMBER:CallerTasks
SMEMBERS Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO:State:COMPENSATED:CallerTasks
SMEMBERS Saga:activate_prepaid_sim:RvMwvxIESAoxMrPO:CONTRACT_TAKEOVER:CallerTasks
## Outputs 8 (in the same order as the queries):
1) "START"
1) "CONTRACT_TAKEOVER"
1) "LIBERATE_NUMBER"
1) "RESERVE_NUMBER"

## Queries 9:
KEYS Saga:activate_prepaid_sim:Task:RESERVE_NUMBER:RequiredTasksSet:*
KEYS Saga:activate_prepaid_sim:Task:LIBERATE_NUMBER:RequiredTasksSet:*
KEYS Saga:activate_prepaid_sim:State:COMPENSATED:RequiredTasksSet:*
KEYS Saga:activate_prepaid_sim:Task:CONTRACT_TAKEOVER:RequiredTasksSet:*
## Outputs 9:
```

```
1) "Saga:activate_prepaid_sim:Task:RESERVE_NUMBER:RequiredTasksSet:0"
1) "Saga:activate_prepaid_sim:Task:LIBERATE_NUMBER:RequiredTasksSet:0"
1) "Saga:activate_prepaid_sim:State:COMPENSATED:RequiredTasksSet:0"
1) "Saga:activate_prepaid_sim:Task:CONTRACT_TAKEOVER:RequiredTasksSet:0"

## Queries 10:
SMEMBERS Saga:activate_prepaid_sim:Task:RESERVE_NUMBER:RequiredTasksSet:0
SMEMBERS Saga:activate_prepaid_sim:Task:LIBERATE_NUMBER:RequiredTasksSet:0
SMEMBERS Saga:activate_prepaid_sim:State:COMPENSATED:RequiredTasksSet:0
SMEMBERS Saga:activate_prepaid_sim:Task:CONTRACT_TAKEOVER:RequiredTasksSet:0
## Outputs 10:
1) "START"
1) "CONTRACT_TAKEOVER"
1) "LIBERATE_NUMBER"
1) "RESERVE_NUMBER"

### Note: The entries of Output 8 are contained in Output 10.
```

Listing B.6: PoC's behaviour validation for the unsuccessful execution of the activate prepaid SIM card saga.

B.2.7 1st Randomly Generated Saga

```
# Context Creation Validation
## Query 1:
EXISTS Saga:random:W6nQtvzhEY69xR7h
## Output 1:
(integer) 1

## Query 2:
EXISTS Saga:random:W6nQtvzhEY69xR7h:state
## Output 2:
(integer) 1

## Query 3:
KEYS Saga:random:W6nQtvzhEY69xR7h:*:CallerTasks
## Output 3:
3) "Saga:random:W6nQtvzhEY69xR7h:AD:CallerTasks"
4) "Saga:random:W6nQtvzhEY69xR7h:E:CallerTasks"
5) "Saga:random:W6nQtvzhEY69xR7h:J:CallerTasks"
6) "Saga:random:W6nQtvzhEY69xR7h:AF:CallerTasks"
7) "Saga:random:W6nQtvzhEY69xR7h:AC:CallerTasks"
8) "Saga:random:W6nQtvzhEY69xR7h:AA:CallerTasks"
9) "Saga:random:W6nQtvzhEY69xR7h:M:CallerTasks"
10) "Saga:random:W6nQtvzhEY69xR7h:State:SUCCESS:CallerTasks"
11) "Saga:random:W6nQtvzhEY69xR7h:C:CallerTasks"
12) "Saga:random:W6nQtvzhEY69xR7h:AG:CallerTasks"
```

```
13) "Saga:random:W6nQtvzhEY69xR7h:N:CallerTasks"
14) "Saga:random:W6nQtvzhEY69xR7h:H:CallerTasks"
15) "Saga:random:W6nQtvzhEY69xR7h:F:CallerTasks"
16) "Saga:random:W6nQtvzhEY69xR7h:AH:CallerTasks"
17) "Saga:random:W6nQtvzhEY69xR7h:G:CallerTasks"
18) "Saga:random:W6nQtvzhEY69xR7h:O:CallerTasks"
19) "Saga:random:W6nQtvzhEY69xR7h:Q:CallerTasks"
20) "Saga:random:W6nQtvzhEY69xR7h:D:CallerTasks"
21) "Saga:random:W6nQtvzhEY69xR7h:V:CallerTasks"
22) "Saga:random:W6nQtvzhEY69xR7h:Z:CallerTasks"
23) "Saga:random:W6nQtvzhEY69xR7h:AE:CallerTasks"
24) "Saga:random:W6nQtvzhEY69xR7h:X:CallerTasks"
25) "Saga:random:W6nQtvzhEY69xR7h:Y:CallerTasks"
26) "Saga:random:W6nQtvzhEY69xR7h:AB:CallerTasks"
27) "Saga:random:W6nQtvzhEY69xR7h:L:CallerTasks"
28) "Saga:random:W6nQtvzhEY69xR7h:I:CallerTasks"
29) "Saga:random:W6nQtvzhEY69xR7h:W:CallerTasks"
30) "Saga:random:W6nQtvzhEY69xR7h:S:CallerTasks"
31) "Saga:random:W6nQtvzhEY69xR7h:A:CallerTasks"
32) "Saga:random:W6nQtvzhEY69xR7h:U:CallerTasks"
33) "Saga:random:W6nQtvzhEY69xR7h:P:CallerTasks"
34) "Saga:random:W6nQtvzhEY69xR7h:B:CallerTasks"

# Tasks Creation Validation
## Query 4:
GRAPH.QUERY Saga:random "MATCH p = (s1:State {name:'START'})-[:success|:failure
    *]->(S2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.name <> 'START'
    AND n.name <> 'SUCCESS' | n.name]"
## Output 4:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
2) 1) 1) "[A, C, K, AD]"
   2) 1) "[A, C, K, W]"
   3) 1) "[A, C, E, AE]"
   4) 1) "[A, C, E, AA, P, Z]"
   5) 1) "[A, C, E, AA, P, X]"
   6) 1) "[A, C, E, AA, P, U, AH]"
   7) 1) "[A, C, E, L, M, S, AB, AC, AG]"
   8) 1) "[A, C, E, H, I, AF]"
   9) 1) "[A, C, E, H, I, J, O, Q, Y]"
   10) 1) "[A, C, E, F, P, Z]"
   11) 1) "[A, C, E, F, P, X]"
   12) 1) "[A, C, E, F, P, U, AH]"
   13) 1) "[A, C, D, V, X]"
   14) 1) "[A, C, D, G, N, V, X]"
   15) 1) "[A, C, D, G, N, T, Y]"
   16) 1) "[A, B, C, K, AD]"
   17) 1) "[A, B, C, K, W]"
   18) 1) "[A, B, C, E, AE]"
   19) 1) "[A, B, C, E, AA, P, Z]"
```

```
   20) 1) "[A, B, C, E, AA, P, X]"
   21) 1) "[A, B, C, E, AA, P, U, AH]"
   22) 1) "[A, B, C, E, L, M, S, AB, AC, AG]"
   23) 1) "[A, B, C, E, H, I, AF]"
   24) 1) "[A, B, C, E, H, I, J, O, Q, Y]"
   25) 1) "[A, B, C, E, F, P, Z]"
   26) 1) "[A, B, C, E, F, P, X]"
   27) 1) "[A, B, C, E, F, P, U, AH]"
   28) 1) "[A, B, C, D, V, X]"
   29) 1) "[A, B, C, D, G, N, V, X]"
   30) 1) "[A, B, C, D, G, N, T, Y]"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 1.537589 milliseconds"

## Query 5:
GRAPH.QUERY Saga:random:W6nQtvzhEY69xR7h "MATCH p = (s1:State {name:'START'})
    -[*]->(s2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.name <> 'START'
     AND n.name <> 'SUCCESS' | n.name]"
## Output 5:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
2) 1) 1) "[A, C, K, AD]"
   2) 1) "[A, C, K, W]"
   3) 1) "[A, C, E, AE]"
   4) 1) "[A, C, E, AA, P, Z]"
   5) 1) "[A, C, E, AA, P, X]"
   6) 1) "[A, C, E, AA, P, U, AH]"
   7) 1) "[A, C, E, L, M, S, AB, AC, AG]"
   8) 1) "[A, C, E, H, I, AF]"
   9) 1) "[A, C, E, H, I, J, O, Q, Y]"
   10) 1) "[A, C, E, F, P, Z]"
   11) 1) "[A, C, E, F, P, X]"
   12) 1) "[A, C, E, F, P, U, AH]"
   13) 1) "[A, C, D, V, X]"
   14) 1) "[A, C, D, G, N, T, Y]"
   15) 1) "[A, C, D, G, N, V, X]"
   16) 1) "[A, B, C, K, AD]"
   17) 1) "[A, B, C, K, W]"
   18) 1) "[A, B, C, E, AE]"
   19) 1) "[A, B, C, E, AA, P, Z]"
   20) 1) "[A, B, C, E, AA, P, X]"
   21) 1) "[A, B, C, E, AA, P, U, AH]"
   22) 1) "[A, B, C, E, L, M, S, AB, AC, AG]"
   23) 1) "[A, B, C, E, H, I, AF]"
   24) 1) "[A, B, C, E, H, I, J, O, Q, Y]"
   25) 1) "[A, B, C, E, F, P, Z]"
   26) 1) "[A, B, C, E, F, P, X]"
   27) 1) "[A, B, C, E, F, P, U, AH]"
   28) 1) "[A, B, C, D, V, X]"
   29) 1) "[A, B, C, D, G, N, T, Y]"
```

```
   30) 1) "[A, B, C, D, G, N, V, X]"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 1.580369 milliseconds"

### Note: Output 5 is present in Output 4.

# Context Final State Validation
## Query 6:
GET Saga:random:W6nQtvzhEY69xR7h:state
## Output 6:
"SUCCESS"

# Tasks Final State Validation
## Query 7:
GRAPH.QUERY Saga:random:W6nQtvzhEY69xR7h "MATCH (t:Task) RETURN t.name as taskName,
     t.technical_state, t.business_state"
## Output 7:
1) 1) "taskName"
   2) "t.technical_state"
   3) "t.business_state"
2) 1) 1) "A"
      2) "ROUTED"
      3) "SUCCESS"
   2) 1) "B"
      2) "ROUTED"
      3) "SUCCESS"
   3) 1) "C"
      2) "ROUTED"
      3) "SUCCESS"
   4) 1) "D"
      2) "ROUTED"
      3) "SUCCESS"
   5) 1) "E"
      2) "ROUTED"
      3) "SUCCESS"
   6) 1) "K"
      2) "ROUTED"
      3) "SUCCESS"
   7) 1) "G"
      2) "ROUTED"
      3) "SUCCESS"
   8) 1) "V"
      2) "ROUTED"
      3) "SUCCESS"
   9) 1) "F"
      2) "ROUTED"
      3) "SUCCESS"
   10) 1) "H"
      2) "ROUTED"
```

```
      3) "SUCCESS"
   11) 1) "L"
      2) "ROUTED"
      3) "SUCCESS"
   12) 1) "AA"
      2) "ROUTED"
      3) "SUCCESS"
   13) 1) "AE"
      2) "ROUTED"
      3) "SUCCESS"
   14) 1) "W"
      2) "ROUTED"
      3) "SUCCESS"
   15) 1) "AD"
      2) "ROUTED"
      3) "SUCCESS"
   16) 1) "N"
      2) "ROUTED"
      3) "SUCCESS"
   17) 1) "P"
      2) "ROUTED"
      3) "SUCCESS"
   18) 1) "I"
      2) "ROUTED"
      3) "SUCCESS"
   19) 1) "M"
      2) "ROUTED"
      3) "SUCCESS"
   20) 1) "T"
      2) "ROUTED"
      3) "SUCCESS"
   21) 1) "J"
      2) "ROUTED"
      3) "SUCCESS"
   22) 1) "AF"
      2) "ROUTED"
      3) "SUCCESS"
   23) 1) "S"
      2) "ROUTED"
      3) "SUCCESS"
   24) 1) "U"
      2) "ROUTED"
      3) "SUCCESS"
   25) 1) "X"
      2) "ROUTED"
      3) "SUCCESS"
   26) 1) "Z"
      2) "ROUTED"
      3) "SUCCESS"
```

```
   27) 1) "Y"
       2) "ROUTED"
       3) "SUCCESS"
   28) 1) "O"
       2) "ROUTED"
       3) "SUCCESS"
   29) 1) "AB"
       2) "ROUTED"
       3) "SUCCESS"
   30) 1) "AH"
       2) "ROUTED"
       3) "SUCCESS"
   31) 1) "Q"
       2) "ROUTED"
       3) "SUCCESS"
   32) 1) "AC"
       2) "ROUTED"
       3) "SUCCESS"
   33) 1) "AG"
       2) "ROUTED"
       3) "SUCCESS"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 0.590550 milliseconds"

# Caller Tasks Registrations Validation
### Note: The keys of the context precedence map are available in output 3.
## Queries 8:
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:AD:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:E:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:J:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:AF:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:AC:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:AA:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:M:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:State:SUCCESS:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:C:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:AG:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:N:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:H:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:F:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:AH:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:G:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:O:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:Q:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:D:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:V:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:Z:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:AE:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:X:CallerTasks
```

```
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:Y:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:AB:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:L:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:I:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:W:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:S:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:A:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:U:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:P:CallerTasks
SMEMBERS Saga:random:W6nQtvzhEY69xR7h:B:CallerTasks
## Outputs 8 (in the same order as the queries):
1) "K"
1) "C"
1) "I"
1) "I"
1) "AB"
1) "E"
1) "L"
1) "W"
2) "X"
3) "Z"
4) "AF"
5) "AD"
6) "Y"
7) "AH"
8) "AG"
9) "AE"
1) "B"
2) "A"
1) "AC"
1) "G"
1) "E"
1) "E"
1) "U"
1) "D"
1) "J"
1) "O"
1) "C"
1) "N"
2) "D"
1) "P"
1) "E"
1) "P"
2) "V"
1) "Q"
2) "T"
1) "S"
1) "E"
1) "H"
```

```
1) "K"
1) "M"
1) "START"
1) "P"
1) "F"
2) "AA"
1) "A"

## Queries 9:
KEYS Saga:random:Task:AD:RequiredTasksSet:*
KEYS Saga:random:Task:E:RequiredTasksSet:*
KEYS Saga:random:Task:J:RequiredTasksSet:*
KEYS Saga:random:Task:AF:RequiredTasksSet:*
KEYS Saga:random:Task:AC:RequiredTasksSet:*
KEYS Saga:random:Task:AA:RequiredTasksSet:*
KEYS Saga:random:Task:M:RequiredTasksSet:*
KEYS Saga:random:State:SUCCESS:RequiredTasksSet:*
KEYS Saga:random:Task:C:RequiredTasksSet:*
KEYS Saga:random:Task:AG:RequiredTasksSet:*
KEYS Saga:random:Task:N:RequiredTasksSet:*
KEYS Saga:random:Task:H:RequiredTasksSet:*
KEYS Saga:random:Task:F:RequiredTasksSet:*
KEYS Saga:random:Task:AH:RequiredTasksSet:*
KEYS Saga:random:Task:G:RequiredTasksSet:*
KEYS Saga:random:Task:O:RequiredTasksSet:*
KEYS Saga:random:Task:Q:RequiredTasksSet:*
KEYS Saga:random:Task:D:RequiredTasksSet:*
KEYS Saga:random:Task:V:RequiredTasksSet:*
KEYS Saga:random:Task:Z:RequiredTasksSet:*
KEYS Saga:random:Task:AE:RequiredTasksSet:*
KEYS Saga:random:Task:X:RequiredTasksSet:*
KEYS Saga:random:Task:Y:RequiredTasksSet:*
KEYS Saga:random:Task:AB:RequiredTasksSet:*
KEYS Saga:random:Task:L:RequiredTasksSet:*
KEYS Saga:random:Task:I:RequiredTasksSet:*
KEYS Saga:random:Task:W:RequiredTasksSet:*
KEYS Saga:random:Task:S:RequiredTasksSet:*
KEYS Saga:random:Task:A:RequiredTasksSet:*
KEYS Saga:random:Task:U:RequiredTasksSet:*
KEYS Saga:random:Task:P:RequiredTasksSet:*
KEYS Saga:random:Task:B:RequiredTasksSet:*
## Outputs 9:
1) "Saga:random:Task:AD:RequiredTasksSet:0"
1) "Saga:random:Task:E:RequiredTasksSet:0"
1) "Saga:random:Task:J:RequiredTasksSet:0"
1) "Saga:random:Task:AF:RequiredTasksSet:0"
1) "Saga:random:Task:AC:RequiredTasksSet:0"
1) "Saga:random:Task:AA:RequiredTasksSet:0"
1) "Saga:random:Task:M:RequiredTasksSet:0"
```

```
1) "Saga:random:State:SUCCESS:RequiredTasksSet:0"
1) "Saga:random:Task:C:RequiredTasksSet:0"
1) "Saga:random:Task:AG:RequiredTasksSet:0"
1) "Saga:random:Task:N:RequiredTasksSet:0"
1) "Saga:random:Task:H:RequiredTasksSet:0"
1) "Saga:random:Task:F:RequiredTasksSet:0"
1) "Saga:random:Task:AH:RequiredTasksSet:0"
1) "Saga:random:Task:G:RequiredTasksSet:0"
1) "Saga:random:Task:O:RequiredTasksSet:0"
1) "Saga:random:Task:Q:RequiredTasksSet:0"
1) "Saga:random:Task:D:RequiredTasksSet:0"
1) "Saga:random:Task:V:RequiredTasksSet:0"
1) "Saga:random:Task:Z:RequiredTasksSet:0"
1) "Saga:random:Task:AE:RequiredTasksSet:0"
1) "Saga:random:Task:X:RequiredTasksSet:0"
1) "Saga:random:Task:Y:RequiredTasksSet:0"
1) "Saga:random:Task:AB:RequiredTasksSet:0"
1) "Saga:random:Task:L:RequiredTasksSet:0"
1) "Saga:random:Task:I:RequiredTasksSet:0"
1) "Saga:random:Task:W:RequiredTasksSet:0"
1) "Saga:random:Task:S:RequiredTasksSet:0"
1) "Saga:random:Task:A:RequiredTasksSet:0"
1) "Saga:random:Task:U:RequiredTasksSet:0"
1) "Saga:random:Task:P:RequiredTasksSet:0"
1) "Saga:random:Task:B:RequiredTasksSet:0"

## Queries 10:
SMEMBERS Saga:random:Task:AD:RequiredTasksSet:0
SMEMBERS Saga:random:Task:E:RequiredTasksSet:0
SMEMBERS Saga:random:Task:J:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AF:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AC:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AA:RequiredTasksSet:0
SMEMBERS Saga:random:Task:M:RequiredTasksSet:0
SMEMBERS Saga:random:State:SUCCESS:RequiredTasksSet:0
SMEMBERS Saga:random:Task:C:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AG:RequiredTasksSet:0
SMEMBERS Saga:random:Task:N:RequiredTasksSet:0
SMEMBERS Saga:random:Task:H:RequiredTasksSet:0
SMEMBERS Saga:random:Task:F:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AH:RequiredTasksSet:0
SMEMBERS Saga:random:Task:G:RequiredTasksSet:0
SMEMBERS Saga:random:Task:O:RequiredTasksSet:0
SMEMBERS Saga:random:Task:Q:RequiredTasksSet:0
SMEMBERS Saga:random:Task:D:RequiredTasksSet:0
SMEMBERS Saga:random:Task:V:RequiredTasksSet:0
SMEMBERS Saga:random:Task:Z:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AE:RequiredTasksSet:0
SMEMBERS Saga:random:Task:X:RequiredTasksSet:0
```

```
SMEMBERS Saga:random:Task:Y:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AB:RequiredTasksSet:0
SMEMBERS Saga:random:Task:L:RequiredTasksSet:0
SMEMBERS Saga:random:Task:I:RequiredTasksSet:0
SMEMBERS Saga:random:Task:W:RequiredTasksSet:0
SMEMBERS Saga:random:Task:S:RequiredTasksSet:0
SMEMBERS Saga:random:Task:A:RequiredTasksSet:0
SMEMBERS Saga:random:Task:U:RequiredTasksSet:0
SMEMBERS Saga:random:Task:P:RequiredTasksSet:0
SMEMBERS Saga:random:Task:B:RequiredTasksSet:0
## Outputs 10:
1) "K"
1) "C"
1) "I"
1) "I"
1) "AB"
1) "E"
1) "L"
1) "AG"
2) "W"
3) "X"
4) "Z"
5) "AE"
6) "AF"
7) "AD"
8) "Y"
9) "AH"
1) "B"
2) "A"
1) "AC"
1) "G"
1) "E"
1) "E"
1) "U"
1) "D"
1) "J"
1) "O"
1) "C"
1) "N"
2) "D"
1) "P"
1) "E"
1) "P"
2) "V"
1) "T"
2) "Q"
1) "S"
1) "E"
1) "H"
```

```
1) "K"
1) "M"
1) "START"
1) "P"
1) "F"
2) "AA"
1) "A"

### Note: Outputs 10 and 8 are equal.
```

Listing B.7: PoC's behaviour validation for the successful execution of a randomly generated saga.

B.2.8 2nd Randomly Generated Saga

```
# Context Creation Validation
## Query 1:
EXISTS Saga:random:AFpdMrhk1cajexGx
## Output 1:
(integer) 1

## Query 2:
EXISTS Saga:random:AFpdMrhk1cajexGx:state
## Output 2:
(integer) 1

## Query 3:
KEYS Saga:random:AFpdMrhk1cajexGx:*:CallerTasks
## Output 3:
1) "Saga:random:AFpdMrhk1cajexGx:K:CallerTasks"
2) "Saga:random:AFpdMrhk1cajexGx:U:CallerTasks"
3) "Saga:random:AFpdMrhk1cajexGx:V:CallerTasks"
4) "Saga:random:AFpdMrhk1cajexGx:E:CallerTasks"
5) "Saga:random:AFpdMrhk1cajexGx:B:CallerTasks"
6) "Saga:random:AFpdMrhk1cajexGx:Z:CallerTasks"
7) "Saga:random:AFpdMrhk1cajexGx:L:CallerTasks"
8) "Saga:random:AFpdMrhk1cajexGx:Y:CallerTasks"
9) "Saga:random:AFpdMrhk1cajexGx:W:CallerTasks"
10) "Saga:random:AFpdMrhk1cajexGx:A:CallerTasks"
11) "Saga:random:AFpdMrhk1cajexGx:N:CallerTasks"
12) "Saga:random:AFpdMrhk1cajexGx:AC:CallerTasks"
13) "Saga:random:AFpdMrhk1cajexGx:G:CallerTasks"
14) "Saga:random:AFpdMrhk1cajexGx:F:CallerTasks"
15) "Saga:random:AFpdMrhk1cajexGx:P:CallerTasks"
16) "Saga:random:AFpdMrhk1cajexGx:Q:CallerTasks"
17) "Saga:random:AFpdMrhk1cajexGx:I:CallerTasks"
18) "Saga:random:AFpdMrhk1cajexGx:AB:CallerTasks"
19) "Saga:random:AFpdMrhk1cajexGx:T:CallerTasks"
```

```
20) "Saga:random:AFpdMrhk1cajexGx:D:CallerTasks"
21) "Saga:random:AFpdMrhk1cajexGx:S:CallerTasks"
22) "Saga:random:AFpdMrhk1cajexGx:AD:CallerTasks"
23) "Saga:random:AFpdMrhk1cajexGx:M:CallerTasks"
24) "Saga:random:AFpdMrhk1cajexGx:State:SUCCESS:CallerTasks"
25) "Saga:random:AFpdMrhk1cajexGx:C:CallerTasks"
26) "Saga:random:AFpdMrhk1cajexGx:H:CallerTasks"
27) "Saga:random:AFpdMrhk1cajexGx:X:CallerTasks"
28) "Saga:random:AFpdMrhk1cajexGx:O:CallerTasks"
29) "Saga:random:AFpdMrhk1cajexGx:J:CallerTasks"
30) "Saga:random:AFpdMrhk1cajexGx:AA:CallerTasks"

# Tasks Creation Validation
## Query 4:
GRAPH.QUERY Saga:random "MATCH p = (s1:State {name:'START'})-[:success|:failure
    *]->(S2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.name <> 'START'
    AND n.name <> 'SUCCESS' | n.name]"
## Output 4:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
2) 1) 1) "[A, W]"
   2) 1) "[A, Q, U, N, Z, AC]"
   3) 1) "[A, M, AA, AB, AD]"
   4) 1) "[A, G, S, M, AA, AB, AD]"
   5) 1) "[A, G, F, Z, AC]"
   6) 1) "[A, G, F, I, X, AA, AB, AD]"
   7) 1) "[A, G, F, I, O, M, AA, AB, AD]"
   8) 1) "[A, B, H, Y, Z, AC]"
   9) 1) "[A, B, G, S, M, AA, AB, AD]"
   10) 1) "[A, B, G, F, Z, AC]"
   11) 1) "[A, B, G, F, I, X, AA, AB, AD]"
   12) 1) "[A, B, G, F, I, O, M, AA, AB, AD]"
   13) 1) "[A, B, F, Z, AC]"
   14) 1) "[A, B, F, I, X, AA, AB, AD]"
   15) 1) "[A, B, F, I, O, M, AA, AB, AD]"
   16) 1) "[A, B, C, E, N, Z, AC]"
   17) 1) "[A, B, C, E, K, V, W]"
   18) 1) "[A, B, C, E, J, W]"
   19) 1) "[A, B, C, D, L, P, T]"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 12.665972 milliseconds"

## Query 5:
GRAPH.QUERY Saga:random:AFpdMrhk1cajexGx "MATCH p = (s1:State {name:'START'})
    -[*]->(s2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.name <> 'START'
     AND n.name <> 'SUCCESS' | n.name]"
## Output 5:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
2) 1) 1) "[A, W]"
   2) 1) "[A, Q, U, N, Z, AC]"
```

```
   3) 1) "[A, M, AA, AB, AD]"
   4) 1) "[A, G, S, M, AA, AB, AD]"
   5) 1) "[A, G, F, I, X, AA, AB, AD]"
   6) 1) "[A, G, F, I, O, M, AA, AB, AD]"
   7) 1) "[A, G, F, Z, AC]"
   8) 1) "[A, B, H, Y, Z, AC]"
   9) 1) "[A, B, F, I, X, AA, AB, AD]"
   10) 1) "[A, B, F, I, O, M, AA, AB, AD]"
   11) 1) "[A, B, F, Z, AC]"
   12) 1) "[A, B, C, E, K, V, W]"
   13) 1) "[A, B, C, E, J, W]"
   14) 1) "[A, B, C, E, N, Z, AC]"
   15) 1) "[A, B, C, D, L, P, T]"
   16) 1) "[A, B, G, S, M, AA, AB, AD]"
   17) 1) "[A, B, G, F, I, X, AA, AB, AD]"
   18) 1) "[A, B, G, F, I, O, M, AA, AB, AD]"
   19) 1) "[A, B, G, F, Z, AC]"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 4.588842 milliseconds"

### Note: Output 5 is present in Output 4.

# Context Final State Validation
## Query 6:
GET Saga:random:AFpdMrhk1cajexGx:state
## Output 6:
"SUCCESS"

# Tasks Final State Validation
## Query 7:
GRAPH.QUERY Saga:random:AFpdMrhk1cajexGx "MATCH (t:Task) RETURN t.name as taskName,
     t.technical_state, t.business_state"
## Output 7:
1) 1) "taskName"
   2) "t.technical_state"
   3) "t.business_state"
2) 1) 1) "A"
      2) "ROUTED"
      3) "SUCCESS"
   2) 1) "B"
      2) "ROUTED"
      3) "SUCCESS"
   3) 1) "G"
      2) "ROUTED"
      3) "SUCCESS"
   4) 1) "M"
      2) "ROUTED"
      3) "SUCCESS"
   5) 1) "Q"
```

```
127        2) "ROUTED"
128        3) "SUCCESS"
129     6) 1) "W"
130        2) "ROUTED"
131        3) "SUCCESS"
132     7) 1) "C"
133        2) "ROUTED"
134        3) "SUCCESS"
135     8) 1) "F"
136        2) "ROUTED"
137        3) "SUCCESS"
138     9) 1) "H"
139        2) "ROUTED"
140        3) "SUCCESS"
141     10) 1) "U"
142        2) "ROUTED"
143        3) "SUCCESS"
144     11) 1) "N"
145        2) "ROUTED"
146        3) "SUCCESS"
147     12) 1) "D"
148        2) "ROUTED"
149        3) "SUCCESS"
150     13) 1) "E"
151        2) "ROUTED"
152        3) "SUCCESS"
153     14) 1) "S"
154        2) "ROUTED"
155        3) "SUCCESS"
156     15) 1) "Y"
157        2) "ROUTED"
158        3) "SUCCESS"
159     16) 1) "Z"
160        2) "ROUTED"
161        3) "SUCCESS"
162     17) 1) "L"
163        2) "ROUTED"
164        3) "SUCCESS"
165     18) 1) "P"
166        2) "ROUTED"
167        3) "SUCCESS"
168     19) 1) "J"
169        2) "ROUTED"
170        3) "SUCCESS"
171     20) 1) "K"
172        2) "ROUTED"
173        3) "SUCCESS"
174     21) 1) "T"
175        2) "ROUTED"
```

```
      3) "SUCCESS"
   22) 1) "V"
      2) "ROUTED"
      3) "SUCCESS"
   23) 1) "I"
      2) "ROUTED"
      3) "SUCCESS"
   24) 1) "O"
      2) "ROUTED"
      3) "SUCCESS"
   25) 1) "X"
      2) "ROUTED"
      3) "SUCCESS"
   26) 1) "AC"
      2) "ROUTED"
      3) "SUCCESS"
   27) 1) "AA"
      2) "ROUTED"
      3) "SUCCESS"
   28) 1) "AB"
      2) "ROUTED"
      3) "SUCCESS"
   29) 1) "AD"
      2) "ROUTED"
      3) "SUCCESS"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 2.380914 milliseconds"

# Caller Tasks Registrations Validation
### Note: The keys of the context precedence map are available in output 3.
## Queries 8:
SMEMBERS Saga:random:AFpdMrhk1cajexGx:K:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:U:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:V:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:E:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:B:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:Z:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:L:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:Y:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:W:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:A:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:N:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:AC:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:G:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:F:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:P:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:Q:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:I:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:AB:CallerTasks
```

```
SMEMBERS Saga:random:AFpdMrhk1cajexGx:T:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:D:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:S:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:AD:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:M:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:State:SUCCESS:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:C:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:H:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:X:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:O:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:J:CallerTasks
SMEMBERS Saga:random:AFpdMrhk1cajexGx:AA:CallerTasks
## Outputs 8 (in the same order as the queries):
1) "E"
1) "Q"
1) "K"
1) "C"
1) "A"
1) "N"
2) "F"
3) "Y"
1) "D"
1) "H"
1) "V"
2) "A"
3) "J"
1) "START"
1) "U"
2) "E"
1) "Z"
1) "A"
2) "B"
1) "B"
2) "G"
1) "L"
1) "A"
1) "F"
1) "AA"
1) "P"
1) "C"
1) "G"
1) "AB"
1) "A"
2) "O"
3) "S"
1) "AD"
2) "W"
3) "T"
4) "AC"
```

```
1) "B"
1) "B"
1) "I"
1) "I"
1) "E"
1) "X"
2) "M"


## Queries 9:
KEYS Saga:random:Task:K:RequiredTasksSet:*
KEYS Saga:random:Task:U:RequiredTasksSet:*
KEYS Saga:random:Task:V:RequiredTasksSet:*
KEYS Saga:random:Task:E:RequiredTasksSet:*
KEYS Saga:random:Task:B:RequiredTasksSet:*
KEYS Saga:random:Task:Z:RequiredTasksSet:*
KEYS Saga:random:Task:L:RequiredTasksSet:*
KEYS Saga:random:Task:Y:RequiredTasksSet:*
KEYS Saga:random:Task:W:RequiredTasksSet:*
KEYS Saga:random:Task:A:RequiredTasksSet:*
KEYS Saga:random:Task:N:RequiredTasksSet:*
KEYS Saga:random:Task:AC:RequiredTasksSet:*
KEYS Saga:random:Task:G:RequiredTasksSet:*
KEYS Saga:random:Task:F:RequiredTasksSet:*
KEYS Saga:random:Task:P:RequiredTasksSet:*
KEYS Saga:random:Task:Q:RequiredTasksSet:*
KEYS Saga:random:Task:I:RequiredTasksSet:*
KEYS Saga:random:Task:AB:RequiredTasksSet:*
KEYS Saga:random:Task:T:RequiredTasksSet:*
KEYS Saga:random:Task:D:RequiredTasksSet:*
KEYS Saga:random:Task:S:RequiredTasksSet:*
KEYS Saga:random:Task:AD:RequiredTasksSet:*
KEYS Saga:random:Task:M:RequiredTasksSet:*
KEYS Saga:random:State:SUCCESS:RequiredTasksSet:*
KEYS Saga:random:Task:C:RequiredTasksSet:*
KEYS Saga:random:Task:H:RequiredTasksSet:*
KEYS Saga:random:Task:X:RequiredTasksSet:*
KEYS Saga:random:Task:O:RequiredTasksSet:*
KEYS Saga:random:Task:J:RequiredTasksSet:*
KEYS Saga:random:Task:AA:RequiredTasksSet:*
## Outputs 9:
1) "Saga:random:Task:K:RequiredTasksSet:0"
1) "Saga:random:Task:U:RequiredTasksSet:0"
1) "Saga:random:Task:V:RequiredTasksSet:0"
1) "Saga:random:Task:E:RequiredTasksSet:0"
1) "Saga:random:Task:B:RequiredTasksSet:0"
1) "Saga:random:Task:Z:RequiredTasksSet:0"
1) "Saga:random:Task:L:RequiredTasksSet:0"
1) "Saga:random:Task:Y:RequiredTasksSet:0"
```

```
1) "Saga:random:Task:W:RequiredTasksSet:0"
1) "Saga:random:Task:A:RequiredTasksSet:0"
1) "Saga:random:Task:N:RequiredTasksSet:0"
1) "Saga:random:Task:AC:RequiredTasksSet:0"
1) "Saga:random:Task:G:RequiredTasksSet:0"
1) "Saga:random:Task:F:RequiredTasksSet:0"
1) "Saga:random:Task:P:RequiredTasksSet:0"
1) "Saga:random:Task:Q:RequiredTasksSet:0"
1) "Saga:random:Task:I:RequiredTasksSet:0"
1) "Saga:random:Task:AB:RequiredTasksSet:0"
1) "Saga:random:Task:T:RequiredTasksSet:0"
1) "Saga:random:Task:D:RequiredTasksSet:0"
1) "Saga:random:Task:S:RequiredTasksSet:0"
1) "Saga:random:Task:AD:RequiredTasksSet:0"
1) "Saga:random:Task:M:RequiredTasksSet:0"
1) "Saga:random:Saga:random:State:SUCCESS:RequiredTasksSet:0"
1) "Saga:random:Task:C:RequiredTasksSet:0"
1) "Saga:random:Task:H:RequiredTasksSet:0"
1) "Saga:random:Task:X:RequiredTasksSet:0"
1) "Saga:random:Task:O:RequiredTasksSet:0"
1) "Saga:random:Task:J:RequiredTasksSet:0"
1) "Saga:random:Task:AA:RequiredTasksSet:0"

## Queries 10:
SMEMBERS Saga:random:Task:K:RequiredTasksSet:0
SMEMBERS Saga:random:Task:U:RequiredTasksSet:0
SMEMBERS Saga:random:Task:V:RequiredTasksSet:0
SMEMBERS Saga:random:Task:E:RequiredTasksSet:0
SMEMBERS Saga:random:Task:B:RequiredTasksSet:0
SMEMBERS Saga:random:Task:Z:RequiredTasksSet:0
SMEMBERS Saga:random:Task:L:RequiredTasksSet:0
SMEMBERS Saga:random:Task:Y:RequiredTasksSet:0
SMEMBERS Saga:random:Task:W:RequiredTasksSet:0
SMEMBERS Saga:random:Task:A:RequiredTasksSet:0
SMEMBERS Saga:random:Task:N:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AC:RequiredTasksSet:0
SMEMBERS Saga:random:Task:G:RequiredTasksSet:0
SMEMBERS Saga:random:Task:F:RequiredTasksSet:0
SMEMBERS Saga:random:Task:P:RequiredTasksSet:0
SMEMBERS Saga:random:Task:Q:RequiredTasksSet:0
SMEMBERS Saga:random:Task:I:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AB:RequiredTasksSet:0
SMEMBERS Saga:random:Task:T:RequiredTasksSet:0
SMEMBERS Saga:random:Task:D:RequiredTasksSet:0
SMEMBERS Saga:random:Task:S:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AD:RequiredTasksSet:0
SMEMBERS Saga:random:Task:M:RequiredTasksSet:0
SMEMBERS Saga:random:State:SUCCESS:RequiredTasksSet:0
SMEMBERS Saga:random:Task:C:RequiredTasksSet:0
```

```
SMEMBERS Saga:random:Task:H:RequiredTasksSet:0
SMEMBERS Saga:random:Task:X:RequiredTasksSet:0
SMEMBERS Saga:random:Task:O:RequiredTasksSet:0
SMEMBERS Saga:random:Task:J:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AA:RequiredTasksSet:0
## Outputs 10:
1) "E"
1) "Q"
1) "K"
1) "C"
1) "A"
1) "N"
2) "Y"
3) "F"
1) "D"
1) "H"
1) "V"
2) "A"
3) "J"
1) "START"
1) "U"
2) "E"
1) "Z"
1) "A"
2) "B"
1) "B"
2) "G"
1) "L"
1) "A"
1) "F"
1) "AA"
1) "P"
1) "C"
1) "G"
1) "AB"
1) "A"
2) "S"
3) "O"
1) "AD"
2) "W"
3) "T"
4) "AC"
1) "B"
1) "B"
1) "I"
1) "I"
1) "E"
1) "X"
2) "M"
```

```
421
422 ### Note: Outputs 10 and 8 are equal.
```

Listing B.8: PoC's behaviour validation for the successful execution of a randomly generated saga.

B.2.9 3rd Randomly Generated Saga

```
1  # Context Creation Validation
2  ## Query 1:
3  EXISTS Saga:random:o6Ka5qbOhvS5R6C7
4  ## Output 1:
5  (integer) 1
6
7  ## Query 2:
8  EXISTS Saga:random:o6Ka5qbOhvS5R6C7:state
9  ## Output 2:
10 (integer) 1
11
12 ## Query 3:
13 KEYS Saga:random:o6Ka5qbOhvS5R6C7:*:CallerTasks
14 ## Output 3:
15 1) "Saga:random:o6Ka5qbOhvS5R6C7:State:SUCCESS:CallerTasks"
16 2) "Saga:random:o6Ka5qbOhvS5R6C7:V:CallerTasks"
17 3) "Saga:random:o6Ka5qbOhvS5R6C7:N:CallerTasks"
18 4) "Saga:random:o6Ka5qbOhvS5R6C7:B:CallerTasks"
19 5) "Saga:random:o6Ka5qbOhvS5R6C7:AC:CallerTasks"
20 6) "Saga:random:o6Ka5qbOhvS5R6C7:X:CallerTasks"
21 7) "Saga:random:o6Ka5qbOhvS5R6C7:C:CallerTasks"
22 8) "Saga:random:o6Ka5qbOhvS5R6C7:G:CallerTasks"
23 9) "Saga:random:o6Ka5qbOhvS5R6C7:AF:CallerTasks"
24 10) "Saga:random:o6Ka5qbOhvS5R6C7:H:CallerTasks"
25 11) "Saga:random:o6Ka5qbOhvS5R6C7:T:CallerTasks"
26 12) "Saga:random:o6Ka5qbOhvS5R6C7:AB:CallerTasks"
27 13) "Saga:random:o6Ka5qbOhvS5R6C7:AD:CallerTasks"
28 14) "Saga:random:o6Ka5qbOhvS5R6C7:D:CallerTasks"
29 15) "Saga:random:o6Ka5qbOhvS5R6C7:K:CallerTasks"
30 16) "Saga:random:o6Ka5qbOhvS5R6C7:I:CallerTasks"
31 17) "Saga:random:o6Ka5qbOhvS5R6C7:AA:CallerTasks"
32 18) "Saga:random:o6Ka5qbOhvS5R6C7:P:CallerTasks"
33 19) "Saga:random:o6Ka5qbOhvS5R6C7:F:CallerTasks"
34 20) "Saga:random:o6Ka5qbOhvS5R6C7:Q:CallerTasks"
35 21) "Saga:random:o6Ka5qbOhvS5R6C7:A:CallerTasks"
36 22) "Saga:random:o6Ka5qbOhvS5R6C7:AG:CallerTasks"
37 23) "Saga:random:o6Ka5qbOhvS5R6C7:AI:CallerTasks"
38 24) "Saga:random:o6Ka5qbOhvS5R6C7:Z:CallerTasks"
39 25) "Saga:random:o6Ka5qbOhvS5R6C7:L:CallerTasks"
40 26) "Saga:random:o6Ka5qbOhvS5R6C7:E:CallerTasks"
```

```
27) "Saga:random:o6Ka5qbOhvS5R6C7:AE:CallerTasks"
28) "Saga:random:o6Ka5qbOhvS5R6C7:Y:CallerTasks"
29) "Saga:random:o6Ka5qbOhvS5R6C7:W:CallerTasks"
30) "Saga:random:o6Ka5qbOhvS5R6C7:U:CallerTasks"
31) "Saga:random:o6Ka5qbOhvS5R6C7:S:CallerTasks"
32) "Saga:random:o6Ka5qbOhvS5R6C7:M:CallerTasks"
33) "Saga:random:o6Ka5qbOhvS5R6C7:J:CallerTasks"
34) "Saga:random:o6Ka5qbOhvS5R6C7:AH:CallerTasks"
35) "Saga:random:o6Ka5qbOhvS5R6C7:O:CallerTasks"

# Tasks Creation Validation
## Query 4:
GRAPH.QUERY Saga:random "MATCH p = (s1:State {name:'START'})-[:success|:failure
    *]->(S2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.name <> 'START'
    AND n.name <> 'SUCCESS' | n.name]"
## Output 4:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
2) 1) 1) "[A, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   2) 1) "[A, D, W, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   3) 1) "[A, D, P, N, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   4) 1) "[A, D, H, O, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   5) 1) "[A, D, H, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   6) 1) "[A, D, H, I, M, U, V, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   7) 1) "[A, B, C, G, N, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   8) 1) "[A, B, C, G, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   9) 1) "[A, B, C, E, F, W, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   10) 1) "[A, B, C, E, F, J, S, D, W, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG,
       AH, AI]"
   11) 1) "[A, B, C, E, F, J, S, D, P, N, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG,
       AH, AI]"
   12) 1) "[A, B, C, E, F, J, S, D, H, O, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG
       , AH, AI]"
   13) 1) "[A, B, C, E, F, J, S, D, H, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]
       "
   14) 1) "[A, B, C, E, F, J, S, D, H, I, M, U, V, T, K, Y, Z, AA, AB, AC, AD, AE,
       AF, AG, AH, AI]"
   15) 1) "[A, B, C, E, F, J, Q, O, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH,
       AI]"
   16) 1) "[A, B, C, E, F, J, L, Q, O, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG,
       AH, AI]"
   17) 1) "[A, B, C, E, F, J, L, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 1.070202 milliseconds"

## Query 5:
GRAPH.QUERY Saga:random:o6Ka5qbOhvS5R6C7 "MATCH p = (s1:State {name:'START'})
    -[*]->(s2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.name <> 'START'
     AND n.name <> 'SUCCESS' | n.name]"
## Output 5:
```

```
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
2) 1) 1) "[A, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   2) 1) "[A, D, P, N, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   3) 1) "[A, D, H, I, M, U, V, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   4) 1) "[A, D, H, O, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   5) 1) "[A, D, H, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   6) 1) "[A, D, W, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   7) 1) "[A, B, C, G, N, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   8) 1) "[A, B, C, G, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   9) 1) "[A, B, C, E, F, W, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
   10) 1) "[A, B, C, E, F, J, S, D, P, N, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG,
       AH, AI]"
   11) 1) "[A, B, C, E, F, J, S, D, H, I, M, U, V, T, K, Y, Z, AA, AB, AC, AD, AE,
       AF, AG, AH, AI]"
   12) 1) "[A, B, C, E, F, J, S, D, H, O, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG
       , AH, AI]"
   13) 1) "[A, B, C, E, F, J, S, D, H, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]
       "
   14) 1) "[A, B, C, E, F, J, S, D, W, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG,
       AH, AI]"
   15) 1) "[A, B, C, E, F, J, Q, O, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH,
       AI]"
   16) 1) "[A, B, C, E, F, J, L, Q, O, X, T, K, Y, Z, AA, AB, AC, AD, AE, AF, AG,
       AH, AI]"
   17) 1) "[A, B, C, E, F, J, L, K, Y, Z, AA, AB, AC, AD, AE, AF, AG, AH, AI]"
3) 1) "Cached execution: 0"

### Note: Output 5 is present in Output 4.

# Context Final State Validation
## Query 6:
GET Saga:random:o6Ka5qbOhvS5R6C7:state
## Output 6:
"SUCCESS"

# Tasks Final State Validation
## Query 7:
GRAPH.QUERY Saga:random:o6Ka5qbOhvS5R6C7 "MATCH (t:Task) RETURN t.name as taskName,
     t.technical_state, t.business_state"
## Output 7:
1) 1) "taskName"
   2) "t.technical_state"
   3) "t.business_state"
2) 1) 1) "A"
      2) "ROUTED"
      3) "SUCCESS"
   2) 1) "B"
      2) "ROUTED"
      3) "SUCCESS"
```

```
120    3) 1) "D"
121       2) "ROUTED"
122       3) "SUCCESS"
123    4) 1) "T"
124       2) "ROUTED"
125       3) "SUCCESS"
126    5) 1) "C"
127       2) "ROUTED"
128       3) "SUCCESS"
129    6) 1) "E"
130       2) "ROUTED"
131       3) "SUCCESS"
132    7) 1) "G"
133       2) "ROUTED"
134       3) "SUCCESS"
135    8) 1) "F"
136       2) "ROUTED"
137       3) "SUCCESS"
138    9) 1) "K"
139       2) "ROUTED"
140       3) "SUCCESS"
141    10) 1) "N"
142       2) "ROUTED"
143       3) "SUCCESS"
144    11) 1) "J"
145       2) "ROUTED"
146       3) "SUCCESS"
147    12) 1) "W"
148       2) "ROUTED"
149       3) "SUCCESS"
150    13) 1) "L"
151       2) "ROUTED"
152       3) "SUCCESS"
153    14) 1) "Q"
154       2) "ROUTED"
155       3) "SUCCESS"
156    15) 1) "S"
157       2) "ROUTED"
158       3) "SUCCESS"
159    16) 1) "H"
160       2) "ROUTED"
161       3) "SUCCESS"
162    17) 1) "P"
163       2) "ROUTED"
164       3) "SUCCESS"
165    18) 1) "O"
166       2) "ROUTED"
167       3) "SUCCESS"
168    19) 1) "X"
```

```
169         2) "ROUTED"
170         3) "SUCCESS"
171      20) 1) "I"
172         2) "ROUTED"
173         3) "SUCCESS"
174      21) 1) "M"
175         2) "ROUTED"
176         3) "SUCCESS"
177      22) 1) "U"
178         2) "ROUTED"
179         3) "SUCCESS"
180      23) 1) "V"
181         2) "ROUTED"
182         3) "SUCCESS"
183      24) 1) "Y"
184         2) "ROUTED"
185         3) "SUCCESS"
186      25) 1) "Z"
187         2) "ROUTED"
188         3) "SUCCESS"
189      26) 1) "AA"
190         2) "ROUTED"
191         3) "SUCCESS"
192      27) 1) "AB"
193         2) "ROUTED"
194         3) "SUCCESS"
195      28) 1) "AC"
196         2) "ROUTED"
197         3) "SUCCESS"
198      29) 1) "AD"
199         2) "ROUTED"
200         3) "SUCCESS"
201      30) 1) "AE"
202         2) "ROUTED"
203         3) "SUCCESS"
204      31) 1) "AF"
205         2) "ROUTED"
206         3) "SUCCESS"
207      32) 1) "AG"
208         2) "ROUTED"
209         3) "SUCCESS"
210      33) 1) "AH"
211         2) "ROUTED"
212         3) "SUCCESS"
213      34) 1) "AI"
214         2) "ROUTED"
215         3) "SUCCESS"
216   3) 1) "Cached execution: 0"
217      2) "Query internal execution time: 0.249053 milliseconds"
```

```
218
219 # Caller Tasks Registrations Validation
220 ### Note: The keys of the context precedence map are available in output 3.
221 ## Queries 8:
222 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:State:SUCCESS:CallerTasks
223 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:V:CallerTasks
224 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:N:CallerTasks
225 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:B:CallerTasks
226 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:AC:CallerTasks
227 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:X:CallerTasks
228 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:C:CallerTasks
229 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:G:CallerTasks
230 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:AF:CallerTasks
231 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:H:CallerTasks
232 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:T:CallerTasks
233 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:AB:CallerTasks
234 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:AD:CallerTasks
235 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:D:CallerTasks
236 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:K:CallerTasks
237 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:I:CallerTasks
238 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:AA:CallerTasks
239 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:P:CallerTasks
240 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:F:CallerTasks
241 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:Q:CallerTasks
242 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:A:CallerTasks
243 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:AG:CallerTasks
244 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:AI:CallerTasks
245 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:Z:CallerTasks
246 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:L:CallerTasks
247 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:E:CallerTasks
248 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:AE:CallerTasks
249 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:Y:CallerTasks
250 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:W:CallerTasks
251 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:U:CallerTasks
252 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:S:CallerTasks
253 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:M:CallerTasks
254 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:J:CallerTasks
255 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:AH:CallerTasks
256 SMEMBERS Saga:random:o6Ka5qbOhvS5R6C7:O:CallerTasks
257 ## Outputs 8 (in the same order as the queries):
258 1) "AI"
259 1) "U"
260 1) "P"
261 2) "G"
262 1) "A"
263 1) "AB"
264 1) "W"
265 2) "O"
266 1) "B"
```

```
1) "C"
1) "AE"
1) "D"
1) "V"
2) "N"
3) "A"
4) "X"
1) "AA"
1) "AC"
1) "A"
2) "S"
1) "T"
2) "H"
3) "L"
4) "G"
1) "H"
1) "Z"
1) "D"
1) "E"
1) "L"
2) "J"
1) "START"
1) "AF"
1) "AH"
1) "Y"
1) "J"
1) "C"
1) "AD"
1) "K"
1) "D"
2) "F"
1) "M"
1) "J"
1) "I"
1) "F"
1) "AG"
1) "H"
2) "Q"


## Queries 9:
KEYS Saga:random:State:SUCCESS:RequiredTasksSet:*
KEYS Saga:random:Task:V:RequiredTasksSet:*
KEYS Saga:random:Task:N:RequiredTasksSet:*
KEYS Saga:random:Task:B:RequiredTasksSet:*
KEYS Saga:random:Task:AC:RequiredTasksSet:*
KEYS Saga:random:Task:X:RequiredTasksSet:*
KEYS Saga:random:Task:C:RequiredTasksSet:*
KEYS Saga:random:Task:G:RequiredTasksSet:*
```

```
KEYS Saga:random:Task:AF:RequiredTasksSet:*
KEYS Saga:random:Task:H:RequiredTasksSet:*
KEYS Saga:random:Task:T:RequiredTasksSet:*
KEYS Saga:random:Task:AB:RequiredTasksSet:*
KEYS Saga:random:Task:AD:RequiredTasksSet:*
KEYS Saga:random:Task:D:RequiredTasksSet:*
KEYS Saga:random:Task:K:RequiredTasksSet:*
KEYS Saga:random:Task:I:RequiredTasksSet:*
KEYS Saga:random:Task:AA:RequiredTasksSet:*
KEYS Saga:random:Task:P:RequiredTasksSet:*
KEYS Saga:random:Task:F:RequiredTasksSet:*
KEYS Saga:random:Task:Q:RequiredTasksSet:*
KEYS Saga:random:Task:A:RequiredTasksSet:*
KEYS Saga:random:Task:AG:RequiredTasksSet:*
KEYS Saga:random:Task:AI:RequiredTasksSet:*
KEYS Saga:random:Task:Z:RequiredTasksSet:*
KEYS Saga:random:Task:L:RequiredTasksSet:*
KEYS Saga:random:Task:E:RequiredTasksSet:*
KEYS Saga:random:Task:AE:RequiredTasksSet:*
KEYS Saga:random:Task:Y:RequiredTasksSet:*
KEYS Saga:random:Task:W:RequiredTasksSet:*
KEYS Saga:random:Task:U:RequiredTasksSet:*
KEYS Saga:random:Task:S:RequiredTasksSet:*
KEYS Saga:random:Task:M:RequiredTasksSet:*
KEYS Saga:random:Task:J:RequiredTasksSet:*
KEYS Saga:random:Task:AH:RequiredTasksSet:*
KEYS Saga:random:Task:O:RequiredTasksSet:*
## Outputs 9:
1) "Saga:random:State:SUCCESS:RequiredTasksSet:0"
1) "Task:V:RequiredTasksSet:0"
1) "Task:N:RequiredTasksSet:0"
1) "Task:B:RequiredTasksSet:0"
1) "Task:AC:RequiredTasksSet:0"
1) "Task:X:RequiredTasksSet:0"
1) "Task:C:RequiredTasksSet:0"
1) "Task:G:RequiredTasksSet:0"
1) "Task:AF:RequiredTasksSet:0"
1) "Task:H:RequiredTasksSet:0"
1) "Task:T:RequiredTasksSet:0"
1) "Task:AB:RequiredTasksSet:0"
1) "Task:AD:RequiredTasksSet:0"
1) "Task:D:RequiredTasksSet:0"
1) "Task:K:RequiredTasksSet:0"
1) "Task:I:RequiredTasksSet:0"
1) "Task:AA:RequiredTasksSet:0"
1) "Task:P:RequiredTasksSet:0"
1) "Task:F:RequiredTasksSet:0"
1) "Task:Q:RequiredTasksSet:0"
1) "Task:A:RequiredTasksSet:0"
```

```
1) "Task:AG:RequiredTasksSet:0"
1) "Task:AI:RequiredTasksSet:0"
1) "Task:Z:RequiredTasksSet:0"
1) "Task:L:RequiredTasksSet:0"
1) "Task:E:RequiredTasksSet:0"
1) "Task:AE:RequiredTasksSet:0"
1) "Task:Y:RequiredTasksSet:0"
1) "Task:W:RequiredTasksSet:0"
1) "Task:U:RequiredTasksSet:0"
1) "Task:S:RequiredTasksSet:0"
1) "Task:M:RequiredTasksSet:0"
1) "Task:J:RequiredTasksSet:0"
1) "Task:AH:RequiredTasksSet:0"
1) "Task:O:RequiredTasksSet:0"

## Queries 10:
SMEMBERS Saga:random:State:SUCCESS:RequiredTasksSet:0
SMEMBERS Saga:random:Task:V:RequiredTasksSet:0
SMEMBERS Saga:random:Task:N:RequiredTasksSet:0
SMEMBERS Saga:random:Task:B:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AC:RequiredTasksSet:0
SMEMBERS Saga:random:Task:X:RequiredTasksSet:0
SMEMBERS Saga:random:Task:C:RequiredTasksSet:0
SMEMBERS Saga:random:Task:G:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AF:RequiredTasksSet:0
SMEMBERS Saga:random:Task:H:RequiredTasksSet:0
SMEMBERS Saga:random:Task:T:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AB:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AD:RequiredTasksSet:0
SMEMBERS Saga:random:Task:D:RequiredTasksSet:0
SMEMBERS Saga:random:Task:K:RequiredTasksSet:0
SMEMBERS Saga:random:Task:I:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AA:RequiredTasksSet:0
SMEMBERS Saga:random:Task:P:RequiredTasksSet:0
SMEMBERS Saga:random:Task:F:RequiredTasksSet:0
SMEMBERS Saga:random:Task:Q:RequiredTasksSet:0
SMEMBERS Saga:random:Task:A:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AG:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AI:RequiredTasksSet:0
SMEMBERS Saga:random:Task:Z:RequiredTasksSet:0
SMEMBERS Saga:random:Task:L:RequiredTasksSet:0
SMEMBERS Saga:random:Task:E:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AE:RequiredTasksSet:0
SMEMBERS Saga:random:Task:Y:RequiredTasksSet:0
SMEMBERS Saga:random:Task:W:RequiredTasksSet:0
SMEMBERS Saga:random:Task:U:RequiredTasksSet:0
SMEMBERS Saga:random:Task:S:RequiredTasksSet:0
SMEMBERS Saga:random:Task:M:RequiredTasksSet:0
SMEMBERS Saga:random:Task:J:RequiredTasksSet:0
```

```
SMEMBERS Saga:random:Task:AH:RequiredTasksSet:0
SMEMBERS Saga:random:Task:O:RequiredTasksSet:0
## Outputs 10:
1) "AI"
1) "U"
1) "P"
2) "G"
1) "A"
1) "AB"
1) "W"
2) "O"
1) "B"
1) "C"
1) "AE"
1) "D"
1) "V"
2) "N"
3) "A"
4) "X"
1) "AA"
1) "AC"
1) "A"
2) "S"
1) "T"
2) "L"
3) "H"
4) "G"
1) "H"
1) "Z"
1) "D"
1) "E"
1) "L"
2) "J"
1) "START"
1) "AF"
1) "AH"
1) "Y"
1) "J"
1) "C"
1) "AD"
1) "K"
1) "F"
2) "D"
1) "M"
1) "J"
1) "I"
1) "F"
1) "AG"
1) "H"
```

```
463 2) "Q"
464
465 ### Note: Outputs 10 and 8 are equal.
```

Listing B.9: PoC's behaviour validation for the successful execution of a randomly generated saga.

B.2.10 4th Randomly Generated Saga

```
1  # Context Creation Validation
2  ## Query 1:
3  EXISTS Saga:random:pGkkjU0TazXn688q
4  ## Output 1:
5  (integer) 1
6
7  ## Query 2:
8  EXISTS Saga:random:pGkkjU0TazXn688q:state
9  ## Output 2:
10 (integer) 1
11
12 ## Query 3:
13 KEYS Saga:random:pGkkjU0TazXn688q:*:CallerTasks
14 ## Output 3:
15 1) "Saga:random:pGkkjU0TazXn688q:L:CallerTasks"
16 2) "Saga:random:pGkkjU0TazXn688q:AC:CallerTasks"
17 3) "Saga:random:pGkkjU0TazXn688q:I:CallerTasks"
18 4) "Saga:random:pGkkjU0TazXn688q:D:CallerTasks"
19 5) "Saga:random:pGkkjU0TazXn688q:M:CallerTasks"
20 6) "Saga:random:pGkkjU0TazXn688q:V:CallerTasks"
21 7) "Saga:random:pGkkjU0TazXn688q:O:CallerTasks"
22 8) "Saga:random:pGkkjU0TazXn688q:C:CallerTasks"
23 9) "Saga:random:pGkkjU0TazXn688q:AB:CallerTasks"
24 10) "Saga:random:pGkkjU0TazXn688q:H:CallerTasks"
25 11) "Saga:random:pGkkjU0TazXn688q:S:CallerTasks"
26 12) "Saga:random:pGkkjU0TazXn688q:A:CallerTasks"
27 13) "Saga:random:pGkkjU0TazXn688q:AE:CallerTasks"
28 14) "Saga:random:pGkkjU0TazXn688q:Y:CallerTasks"
29 15) "Saga:random:pGkkjU0TazXn688q:State:SUCCESS:CallerTasks"
30 16) "Saga:random:pGkkjU0TazXn688q:AG:CallerTasks"
31 17) "Saga:random:pGkkjU0TazXn688q:Q:CallerTasks"
32 18) "Saga:random:pGkkjU0TazXn688q:AA:CallerTasks"
33 19) "Saga:random:pGkkjU0TazXn688q:E:CallerTasks"
34 20) "Saga:random:pGkkjU0TazXn688q:X:CallerTasks"
35 21) "Saga:random:pGkkjU0TazXn688q:U:CallerTasks"
36 22) "Saga:random:pGkkjU0TazXn688q:AF:CallerTasks"
37 23) "Saga:random:pGkkjU0TazXn688q:W:CallerTasks"
38 24) "Saga:random:pGkkjU0TazXn688q:K:CallerTasks"
39 25) "Saga:random:pGkkjU0TazXn688q:J:CallerTasks"
```

```
26) "Saga:random:pGkkjU0TazXn688q:AJ:CallerTasks"
27) "Saga:random:pGkkjU0TazXn688q:T:CallerTasks"
28) "Saga:random:pGkkjU0TazXn688q:G:CallerTasks"
29) "Saga:random:pGkkjU0TazXn688q:F:CallerTasks"
30) "Saga:random:pGkkjU0TazXn688q:AH:CallerTasks"
31) "Saga:random:pGkkjU0TazXn688q:N:CallerTasks"
32) "Saga:random:pGkkjU0TazXn688q:B:CallerTasks"
33) "Saga:random:pGkkjU0TazXn688q:AI:CallerTasks"
34) "Saga:random:pGkkjU0TazXn688q:Z:CallerTasks"
35) "Saga:random:pGkkjU0TazXn688q:P:CallerTasks"
36) "Saga:random:pGkkjU0TazXn688q:AD:CallerTasks"

# Tasks Creation Validation
## Query 4:
GRAPH.QUERY Saga:random "MATCH p = (s1:State {name:'START'})-[:success|:failure
    *]->(S2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.name <> 'START'
    AND n.name <> 'SUCCESS' | n.name]"
## Output 4:
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
2) 1) 1) "[A, E, M, U, AA, AE, AI]"
   2) 1) "[A, E, M, U, Y, N, AD, AG, AH, AJ]"
   3) 1) "[A, E, M, T, N, AD, AG, AH, AJ]"
   4) 1) "[A, E, M, Q, W, AD, AG, AH, AJ]"
   5) 1) "[A, E, M, Q, W, AC, AE, AI]"
   6) 1) "[A, E, M, Q, W, AB]"
   7) 1) "[A, E, M, P, K, V, X, AD, AG, AH, AJ]"
   8) 1) "[A, E, M, N, AD, AG, AH, AJ]"
   9) 1) "[A, B, D, G, AF]"
   10) 1) "[A, B, D, G, Z, W, AD, AG, AH, AJ]"
   11) 1) "[A, B, D, G, Z, W, AC, AE, AI]"
   12) 1) "[A, B, D, G, Z, W, AB]"
   13) 1) "[A, B, D, G, H, J, K, V, X, AD, AG, AH, AJ]"
   14) 1) "[A, B, D, F, L, M, U, AA, AE, AI]"
   15) 1) "[A, B, D, F, L, M, U, Y, N, AD, AG, AH, AJ]"
   16) 1) "[A, B, D, F, L, M, T, N, AD, AG, AH, AJ]"
   17) 1) "[A, B, D, F, L, M, Q, W, AD, AG, AH, AJ]"
   18) 1) "[A, B, D, F, L, M, Q, W, AC, AE, AI]"
   19) 1) "[A, B, D, F, L, M, Q, W, AB]"
   20) 1) "[A, B, D, F, L, M, P, K, V, X, AD, AG, AH, AJ]"
   21) 1) "[A, B, D, F, L, M, N, AD, AG, AH, AJ]"
   22) 1) "[A, B, C, I, O, S]"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 0.919260 milliseconds"

## Query 5:
GRAPH.QUERY Saga:random:pGkkjU0TazXn688q "MATCH p = (s1:State {name:'START'})
    -[*]->(s2:State {name:'SUCCESS'}) RETURN [n in nodes(p) WHERE n.name <> 'START'
     AND n.name <> 'SUCCESS' | n.name]"
## Output 5:
```

```
1) 1) "[n in nodes(p) WHERE n.name <> 'START' AND n.name <> 'SUCCESS' | n.name]"
2) 1) 1) "[A, E, M, U, AA, AE, AI]"
   2) 1) "[A, E, M, U, Y, N, AD, AG, AH, AJ]"
   3) 1) "[A, E, M, T, N, AD, AG, AH, AJ]"
   4) 1) "[A, E, M, Q, W, AD, AG, AH, AJ]"
   5) 1) "[A, E, M, Q, W, AC, AE, AI]"
   6) 1) "[A, E, M, Q, W, AB]"
   7) 1) "[A, E, M, P, K, V, X, AD, AG, AH, AJ]"
   8) 1) "[A, E, M, N, AD, AG, AH, AJ]"
   9) 1) "[A, B, D, G, AF]"
   10) 1) "[A, B, D, G, Z, W, AD, AG, AH, AJ]"
   11) 1) "[A, B, D, G, Z, W, AC, AE, AI]"
   12) 1) "[A, B, D, G, Z, W, AB]"
   13) 1) "[A, B, D, G, H, J, K, V, X, AD, AG, AH, AJ]"
   14) 1) "[A, B, D, F, L, M, U, AA, AE, AI]"
   15) 1) "[A, B, D, F, L, M, U, Y, N, AD, AG, AH, AJ]"
   16) 1) "[A, B, D, F, L, M, T, N, AD, AG, AH, AJ]"
   17) 1) "[A, B, D, F, L, M, Q, W, AD, AG, AH, AJ]"
   18) 1) "[A, B, D, F, L, M, Q, W, AC, AE, AI]"
   19) 1) "[A, B, D, F, L, M, Q, W, AB]"
   20) 1) "[A, B, D, F, L, M, P, K, V, X, AD, AG, AH, AJ]"
   21) 1) "[A, B, D, F, L, M, N, AD, AG, AH, AJ]"
   22) 1) "[A, B, C, I, O, S]"
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 0.875826 milliseconds"

### Note: Output 5 is present in Output 4.

# Context Final State Validation
## Query 6:
GET Saga:random:pGkkjU0TazXn688q:state
## Output 6:
"SUCCESS"

# Tasks Final State Validation
## Query 7:
GRAPH.QUERY Saga:random:pGkkjU0TazXn688q "MATCH (t:Task) RETURN t.name as taskName,
     t.technical_state, t.business_state"
## Output 7:
1) 1) "taskName"
   2) "t.technical_state"
   3) "t.business_state"
2) 1) 1) "A"
      2) "ROUTED"
      3) "SUCCESS"
   2) 1) "B"
      2) "ROUTED"
      3) "SUCCESS"
   3) 1) "E"
```

```
      2) "ROUTED"
      3) "SUCCESS"
   4) 1) "C"
      2) "ROUTED"
      3) "SUCCESS"
   5) 1) "D"
      2) "ROUTED"
      3) "SUCCESS"
   6) 1) "M"
      2) "ROUTED"
      3) "SUCCESS"
   7) 1) "I"
      2) "ROUTED"
      3) "SUCCESS"
   8) 1) "F"
      2) "ROUTED"
      3) "SUCCESS"
   9) 1) "G"
      2) "ROUTED"
      3) "SUCCESS"
   10) 1) "O"
      2) "ROUTED"
      3) "SUCCESS"
   11) 1) "S"
      2) "ROUTED"
      3) "SUCCESS"
   12) 1) "L"
      2) "ROUTED"
      3) "SUCCESS"
   13) 1) "H"
      2) "ROUTED"
      3) "SUCCESS"
   14) 1) "Z"
      2) "ROUTED"
      3) "SUCCESS"
   15) 1) "AF"
      2) "ROUTED"
      3) "SUCCESS"
   16) 1) "W"
      2) "ROUTED"
      3) "SUCCESS"
   17) 1) "N"
      2) "ROUTED"
      3) "SUCCESS"
   18) 1) "P"
      2) "ROUTED"
      3) "SUCCESS"
   19) 1) "Q"
      2) "ROUTED"
```

```
182        3) "SUCCESS"
183     20) 1) "T"
184        2) "ROUTED"
185        3) "SUCCESS"
186     21) 1) "U"
187        2) "ROUTED"
188        3) "SUCCESS"
189     22) 1) "J"
190        2) "ROUTED"
191        3) "SUCCESS"
192     23) 1) "K"
193        2) "ROUTED"
194        3) "SUCCESS"
195     24) 1) "AB"
196        2) "ROUTED"
197        3) "SUCCESS"
198     25) 1) "AC"
199        2) "ROUTED"
200        3) "SUCCESS"
201     26) 1) "AD"
202        2) "ROUTED"
203        3) "SUCCESS"
204     27) 1) "Y"
205        2) "ROUTED"
206        3) "SUCCESS"
207     28) 1) "AA"
208        2) "ROUTED"
209        3) "SUCCESS"
210     29) 1) "AE"
211        2) "ROUTED"
212        3) "SUCCESS"
213     30) 1) "V"
214        2) "ROUTED"
215        3) "SUCCESS"
216     31) 1) "X"
217        2) "ROUTED"
218        3) "SUCCESS"
219     32) 1) "AI"
220        2) "ROUTED"
221        3) "SUCCESS"
222     33) 1) "AG"
223        2) "ROUTED"
224        3) "SUCCESS"
225     34) 1) "AH"
226        2) "ROUTED"
227        3) "SUCCESS"
228     35) 1) "AJ"
229        2) "ROUTED"
230        3) "SUCCESS"
```

```
3) 1) "Cached execution: 0"
   2) "Query internal execution time: 0.394993 milliseconds"

# Caller Tasks Registrations Validation
### Note: The keys of the context precedence map are available in output 3.
## Queries 8:
SMEMBERS Saga:random:pGkkjU0TazXn688q:L:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:AC:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:I:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:D:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:M:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:V:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:O:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:C:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:AB:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:H:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:S:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:A:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:AE:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:Y:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:State:SUCCESS:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:AG:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:Q:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:AA:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:E:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:X:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:U:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:AF:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:W:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:K:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:J:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:AJ:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:T:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:G:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:F:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:AH:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:N:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:B:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:AI:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:Z:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:P:CallerTasks
SMEMBERS Saga:random:pGkkjU0TazXn688q:AD:CallerTasks
## Outputs 8 (in the same order as the queries):
1) "F"
1) "W"
1) "C"
1) "B"
1) "L"
2) "E"
```

```
1) "K"
1) "I"
1) "B"
1) "W"
1) "G"
1) "O"
1) "START"
1) "AA"
2) "AC"
1) "U"
1) "AI"
2) "AJ"
3) "AF"
4) "AB"
5) "S"
1) "AD"
1) "M"
1) "U"
1) "A"
1) "V"
1) "M"
1) "G"
1) "Z"
2) "Q"
1) "P"
2) "J"
1) "H"
1) "AH"
1) "M"
1) "D"
1) "D"
1) "AG"
1) "T"
2) "Y"
3) "M"
1) "A"
1) "AE"
1) "G"
1) "M"
1) "N"
2) "W"
3) "X"


## Queries 9:
KEYS Saga:random:Task:L:RequiredTasksSet:*
KEYS Saga:random:Task:AC:RequiredTasksSet:*
KEYS Saga:random:Task:I:RequiredTasksSet:*
KEYS Saga:random:Task:D:RequiredTasksSet:*
```

```
KEYS Saga:random:Task:M:RequiredTasksSet:*
KEYS Saga:random:Task:V:RequiredTasksSet:*
KEYS Saga:random:Task:O:RequiredTasksSet:*
KEYS Saga:random:Task:C:RequiredTasksSet:*
KEYS Saga:random:Task:AB:RequiredTasksSet:*
KEYS Saga:random:Task:H:RequiredTasksSet:*
KEYS Saga:random:Task:S:RequiredTasksSet:*
KEYS Saga:random:Task:A:RequiredTasksSet:*
KEYS Saga:random:Task:AE:RequiredTasksSet:*
KEYS Saga:random:Task:Y:RequiredTasksSet:*
KEYS Saga:random:State:SUCCESS:RequiredTasksSet:*
KEYS Saga:random:Task:AG:RequiredTasksSet:*
KEYS Saga:random:Task:Q:RequiredTasksSet:*
KEYS Saga:random:Task:AA:RequiredTasksSet:*
KEYS Saga:random:Task:E:RequiredTasksSet:*
KEYS Saga:random:Task:X:RequiredTasksSet:*
KEYS Saga:random:Task:U:RequiredTasksSet:*
KEYS Saga:random:Task:AF:RequiredTasksSet:*
KEYS Saga:random:Task:W:RequiredTasksSet:*
KEYS Saga:random:Task:K:RequiredTasksSet:*
KEYS Saga:random:Task:J:RequiredTasksSet:*
KEYS Saga:random:Task:AJ:RequiredTasksSet:*
KEYS Saga:random:Task:T:RequiredTasksSet:*
KEYS Saga:random:Task:G:RequiredTasksSet:*
KEYS Saga:random:Task:F:RequiredTasksSet:*
KEYS Saga:random:Task:AH:RequiredTasksSet:*
KEYS Saga:random:Task:N:RequiredTasksSet:*
KEYS Saga:random:Task:B:RequiredTasksSet:*
KEYS Saga:random:Task:AI:RequiredTasksSet:*
KEYS Saga:random:Task:Z:RequiredTasksSet:*
KEYS Saga:random:Task:P:RequiredTasksSet:*
KEYS Saga:random:Task:AD:RequiredTasksSet:*
## Outputs 9:
1) "Saga:random:Task:L:RequiredTasksSet:0"
1) "Saga:random:Task:AC:RequiredTasksSet:0"
1) "Saga:random:Task:I:RequiredTasksSet:0"
1) "Saga:random:Task:D:RequiredTasksSet:0"
1) "Saga:random:Task:M:RequiredTasksSet:0"
1) "Saga:random:Task:V:RequiredTasksSet:0"
1) "Saga:random:Task:O:RequiredTasksSet:0"
1) "Saga:random:Task:C:RequiredTasksSet:0"
1) "Saga:random:Task:AB:RequiredTasksSet:0"
1) "Saga:random:Task:H:RequiredTasksSet:0"
1) "Saga:random:Task:S:RequiredTasksSet:0"
1) "Saga:random:Task:A:RequiredTasksSet:0"
1) "Saga:random:Task:AE:RequiredTasksSet:0"
1) "Saga:random:Task:Y:RequiredTasksSet:0"
1) "Saga:random:State:SUCCESS:RequiredTasksSet:0"
1) "Saga:random:Task:AG:RequiredTasksSet:0"
```

```
1) "Saga:random:Task:Q:RequiredTasksSet:0"
1) "Saga:random:Task:AA:RequiredTasksSet:0"
1) "Saga:random:Task:E:RequiredTasksSet:0"
1) "Saga:random:Task:X:RequiredTasksSet:0"
1) "Saga:random:Task:U:RequiredTasksSet:0"
1) "Saga:random:Task:AF:RequiredTasksSet:0"
1) "Saga:random:Task:W:RequiredTasksSet:0"
1) "Saga:random:Task:K:RequiredTasksSet:0"
1) "Saga:random:Task:J:RequiredTasksSet:0"
1) "Saga:random:Task:AJ:RequiredTasksSet:0"
1) "Saga:random:Task:T:RequiredTasksSet:0"
1) "Saga:random:Task:G:RequiredTasksSet:0"
1) "Saga:random:Task:F:RequiredTasksSet:0"
1) "Saga:random:Task:AH:RequiredTasksSet:0"
1) "Saga:random:Task:N:RequiredTasksSet:0"
1) "Saga:random:Task:B:RequiredTasksSet:0"
1) "Saga:random:Task:AI:RequiredTasksSet:0"
1) "Saga:random:Task:Z:RequiredTasksSet:0"
1) "Saga:random:Task:P:RequiredTasksSet:0"
1) "Saga:random:Task:AD:RequiredTasksSet:0"

## Queries 10:
SMEMBERS Saga:random:Task:L:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AC:RequiredTasksSet:0
SMEMBERS Saga:random:Task:I:RequiredTasksSet:0
SMEMBERS Saga:random:Task:D:RequiredTasksSet:0
SMEMBERS Saga:random:Task:M:RequiredTasksSet:0
SMEMBERS Saga:random:Task:V:RequiredTasksSet:0
SMEMBERS Saga:random:Task:O:RequiredTasksSet:0
SMEMBERS Saga:random:Task:C:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AB:RequiredTasksSet:0
SMEMBERS Saga:random:Task:H:RequiredTasksSet:0
SMEMBERS Saga:random:Task:S:RequiredTasksSet:0
SMEMBERS Saga:random:Task:A:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AE:RequiredTasksSet:0
SMEMBERS Saga:random:Task:Y:RequiredTasksSet:0
SMEMBERS Saga:random:State:SUCCESS:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AG:RequiredTasksSet:0
SMEMBERS Saga:random:Task:Q:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AA:RequiredTasksSet:0
SMEMBERS Saga:random:Task:E:RequiredTasksSet:0
SMEMBERS Saga:random:Task:X:RequiredTasksSet:0
SMEMBERS Saga:random:Task:U:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AF:RequiredTasksSet:0
SMEMBERS Saga:random:Task:W:RequiredTasksSet:0
SMEMBERS Saga:random:Task:K:RequiredTasksSet:0
SMEMBERS Saga:random:Task:J:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AJ:RequiredTasksSet:0
SMEMBERS Saga:random:Task:T:RequiredTasksSet:0
```

```
SMEMBERS Saga:random:Task:G:RequiredTasksSet:0
SMEMBERS Saga:random:Task:F:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AH:RequiredTasksSet:0
SMEMBERS Saga:random:Task:N:RequiredTasksSet:0
SMEMBERS Saga:random:Task:B:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AI:RequiredTasksSet:0
SMEMBERS Saga:random:Task:Z:RequiredTasksSet:0
SMEMBERS Saga:random:Task:P:RequiredTasksSet:0
SMEMBERS Saga:random:Task:AD:RequiredTasksSet:0
## Outputs 10:
1) "F"
1) "W"
1) "C"
1) "B"
1) "L"
2) "E"
1) "K"
1) "I"
1) "B"
1) "W"
1) "G"
1) "O"
1) "START"
1) "AC"
2) "AA"
1) "U"
1) "AI"
2) "AF"
3) "AB"
4) "S"
5) "AJ"
1) "AD"
1) "M"
1) "U"
1) "A"
1) "V"
1) "M"
1) "G"
1) "Z"
2) "Q"
1) "P"
2) "J"
1) "H"
1) "AH"
1) "M"
1) "D"
1) "D"
1) "AG"
1) "T"
```

```
2) "Y"
3) "M"
1) "A"
1) "AE"
1) "G"
1) "M"
1) "W"
2) "N"
3) "X"

### Note: Outputs 10 and 8 are equal.
```

Listing B.10: PoC's behaviour validation for the successful execution of a randomly generated saga.

Appendix C

Paper Submission

A scientific article on this dissertation is going to be submitted for publishing at the ICSOC 2023 [42] conference. For now, its abstract has been submitted, and the full paper will be submitted soon. The following page presents the abstract that was submitted.

A MuleSoft Framework for the Generic Implementation of the Saga Pattern*

João Pinto[1,2] and Sérgio Lopes[2]

[1] University of Porto (FEUP) up201806667@edu.fe.up.pt
[2] Deloitte Portugal {joaocapinto, serlopes}@deloitte.pt

Abstract. This paper, developed in close collaboration with Deloitte, explores the practice of transaction management in the field of systems integration. This field is responsible for designing solutions that enable the communication of systems of various natures. These solutions are typically built using platforms such as Mulesoft that focus on the creation of application networks via Application Programming Interfaces (APIs). The application of integration techniques often results in a distributed system. Distributed systems have the need to carefully manage transactions. The Saga Pattern is a transaction management solution; however, its implementations are often tied to specific use cases, limiting their general applicability. This work will study the viability of a generic implementation of the Saga Pattern in Mulesoft using microservices choreography.

In this work, Shepherd, a Mulesoft framework, is proposed. Shepherd implements the Saga Pattern using microservices choreography in a generic manner, with the capability of adapting to different use cases. The framework uses graph-based representations of sagas for the configuration of its application and its architecture is of distributed nature.

Validations performed on Shepherd revealed that it successfully implements the Saga Pattern and that a significant efficiency gain was caused by the usage of the framework in an asset reutilization activity.

This research concludes that the Shepherd framework has a high potential for applicability in the industry and challenges common approaches to integration development such as API-led Connectivity. Shepherd introduces a novel concept to the field: an API whose business logic is configurable.

Keywords: Mulesoft · Framework · Saga Pattern · Integration · API.

* Supported by Deloitte.