

Saga Pattern Technologies: A Criteria-based Evaluation

Karolin Dürr, Robin Lichtenthäler^a and Guido Wirtz^b

Distributed Systems Group, University of Bamberg, Germany

Keywords: Microservices, Saga Pattern, Workflow Engine, Criteria-based.

Abstract: One challenge in Microservices Architectures is coordinating business workflows between services, for which the Saga pattern is frequently mentioned as a solution in the literature. This work presents a criteria catalog based on which existing technological solutions that help with Saga implementations can be qualitatively evaluated to enable an informed decision between them. It considers criteria relevant for the Saga pattern, microservices characteristics, and for operating a system in production. We use it to evaluate four technological solutions by implementing an exemplary use case. Due to their different origins, the technologies come with varying strengths and weaknesses and as a result no solution is superior. The results can help developers decide which technology to use and provide insights into what to consider when implementing the Saga pattern.

1 INTRODUCTION

The *microservices architectural style*, popularized by companies like Netflix or Amazon, distributes a system into small and autonomous services (Alshuqayran et al., 2016; Dragoni et al., 2017) communicating over the network, each modeled around one business domain and exclusively owning its data storage. Therefore, microservices can be independently deployed, scaled, maintained, and evolved (Newman, 2015; Richardson, 2019). For business workflows involving several microservices, however, coordination between services is required to ensure consistency. Using distributed transactions, for example with the *2-Phase Commit (2PC) protocol*, would be a classical approach (Al-Houmailya and Samaras, 2009; Newman, 2019) in this regard. But with 2PC, the overall availability and scalability of a system is affected (Newman, 2015; Richardson, 2019; Helland, 2016), due to the strict locking requirements (Thomson et al., 2012) combined with the comparatively slow and unreliable network-based communication.

The *Saga pattern* is therefore mentioned frequently (Richardson, 2019; Newman, 2019; Štefanko et al., 2019; Garcia-Molina and Salem, 1987) as a solution. It divides a transaction into multiple local transactions so that locks for included resources do not have to be held until full completion. For workflows involving multiple services, the Saga pattern


therefore aligns better with the microservices characteristic of autonomy (Dürr et al., 2021).


Nevertheless, the Saga pattern involves complexity, because such independent local transactions need to be coordinated and compensation must be possible to take into account different failure scenarios. Therefore, framework support can be helpful and suitable technologies have emerged. This work investigates different existing solutions by comparing their capabilities based on a well-defined criteria catalog. We acknowledge that the Saga pattern is not suitable for every use case, but this work does not provide an evaluation of when it fits and when it does not. Instead, we make the assumption that the Saga pattern has already been identified as suitable for a use case at hand and a technological solution for supporting the implementation needs to be chosen. Therefore, we also consider aspects important when operating a system in production, like monitoring or security. We summarize our aim with the following research questions:

RQ1: What criteria can be used to evaluate and compare existing technologies that support the implementation of the Saga pattern?

RQ2: How do recent technological solutions that support implementing the Saga pattern perform concerning the defined criteria?

In the following, we shortly describe the Saga pattern in section 2, review similar evaluations in section 3, and outline our approach in section 4. We evaluate the technologies based on our criteria catalog in section 5, discuss it in section 6 and conclude in section 7.

^a  <https://orcid.org/0000-0002-9608-619X>

^b  <https://orcid.org/0000-0002-0438-8482>

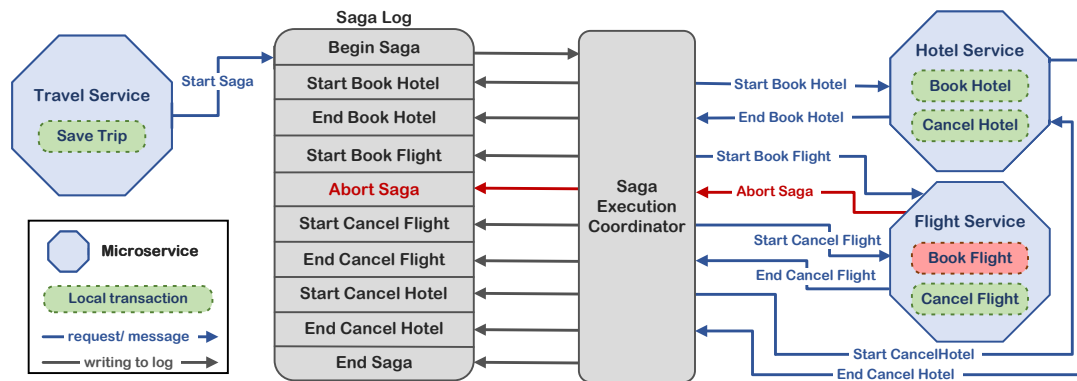


Figure 1: Execution of a Saga's failure scenario based on (Dürr et al., 2021) and ¹.

2 SAGA PATTERN

In 1987 (Garcia-Molina and Salem, 1987) introduced the Saga pattern to manage *long-lived transactions (LLT)*. A Saga designs such a LLT as a sequence of local transactions instead of a single ACID (Vossen, 2009) transaction as in the 2PC protocol.

Therefore, Sagas prevent the need for extended locking of resources and allow updating data in multiple services without using distributed transactions (Richardson, 2019). But it also means that an approach is needed to coordinate these independent services, control each Saga's progress, and resolve inconsistencies eventually through compensation when needed. Coordination of Sagas can be achieved either through choreography (Limón et al., 2018) in a decentralized way or through orchestration where coordination is centralized in a so-called *Saga Execution Coordinator (SEC)*, as proposed by (Garcia-Molina and Salem, 1987). Because orchestration is the approach originally proposed by (Garcia-Molina and Salem, 1987) and it was found to be better suited for complex business contexts by (Bruce and Pereira, 2018), in this paper we focus on the orchestration approach. As shown in fig. 1, the SEC is the orchestrator (Newman, 2019; Bruce and Pereira, 2018) and it is in itself a stateless service that can be realized either within an existing service or as a separate one. The SEC relies on the Saga log, a distributed, persistent log where the status of local transactions of all Saga instances is tracked to control the progress and enable a continuation even if the SEC temporarily fails. For further clarification, we now discuss the example shown in fig. 1 of a trip booking which can be considered as a LLT that includes booking a hotel and a

flight. This example was also used by Catie McCafrey² to introduce the Saga pattern in a microservices architecture including a Travel Service that saves the booking information and initiates the execution, a Hotel Service responsible for hotel bookings and a Flight Service for flight bookings.

Here, the SEC is either part of the Travel Service or an individual service and relies on the services' interfaces to execute Sagas. Upon a trip request, the Travel Service initiates the Saga instance and executes the Save Trip transaction locally. Then, the SEC triggers the hotel booking and waits for a response. Booking a hotel, however, is completely in the responsibility of the Hotel Service which controls its own state and has to ensure consistency through local transactions. Therefore, the SEC relies on the provided functionality without knowing about details of such local transactions. If booking a hotel succeeds, the SEC is informed and triggers the next service to proceed (Richardson, 2019; Limón et al., 2018) until all transactions are completed and thereby the Saga itself is completed. However, in case a local transaction fails like the flight booking in fig. 1, all previously done changes have to be compensated. The most commonly used strategy uses so-called compensating transactions (Richardson, 2019; Garcia-Molina and Salem, 1987). A compensating transaction rolls the previous changes back, either entirely or at least semantically (Newman, 2019; Garcia-Molina and Salem, 1987). Each service participating in a Saga has to guarantee the possibility to compensate state changes introduced by previous transactions. This might also include additional compensations external to the Saga in focus but required due to such state changes. However, this remains the responsibility of the service currently providing compensation.

¹<https://speakerdeck.com/caitiem20/applying-the-saga-pattern?slide=70>

²<https://www.youtube.com/watch?v=xDuwrtwYHu8>

In our example, this means that the Hotel Service has to ensure the compensation of the hotel booking if the flight booking fails and therefore has to be canceled.

3 RELATED WORK

Some previous studies have also assessed existing technologies with regards to the Saga pattern. The study by (Štefanko et al., 2019) analyzed four different Java-based application frameworks: Axon³, Eventuate Event Sourcing⁴ and, like us, Eventuate Tram⁵ and MicroProfile *Long-Running Actions (LRA)*⁶, but based on the Narayana⁷ implementation. Their evaluation focused on assessing the frameworks' support for the Saga pattern, its implementation complexity, and a performance analysis investigating the system's behavior under large load (Štefanko et al., 2019). However, they do not specify their used quality criteria in detail. As a further contribution, they reported discovered problems for each framework. (Dürr et al., 2021) used a solely qualitative analysis to compare two technological solutions: Eventuate Tram and the workflow orchestration engine Netflix Conductor⁸. They consider general Saga execution characteristics, as well as microservices characteristics and challenges for their evaluation criteria, but do not describe their criteria in detail. Our work therefore extends their work by considering additional technological solutions as well as an extended criteria catalog. Besides these two studies, (Fugaro and Vocale, 2019) shortly describe different Saga implementation possibilities in their book and briefly address some of the challenges of each framework, but without an evaluation according to specific criteria. In this context, they mention Axon, Eventuate Event Sourcing, and Eventuate Tram, but focus on an implementation proposal using MicroProfile LRA.

4 METHODOLOGY

While considering the aforementioned evaluations (Dürr et al., 2021; Fugaro and Vocale, 2019; Štefanko et al., 2019) in combination with a literature review, we create a criteria catalog that can be used for a

³<https://docs.axoniq.io/reference-guide/axon-framework/sagas>

⁴<https://eventuate.io/usingeventuate.html>

⁵<https://eventuate.io/abouteventuatetram.html>

⁶<https://microprofile.io/project/eclipse/microprofile-lra>

⁷<https://narayana.io/lra/>

⁸<https://netflix.github.io/conductor/>

technological evaluation. As additional resources we used further literature (Estdale and Georgiadou, 2018; Gaffney, 1981; Cruz et al., 2006; Confino and Laplante, 2010) as well as the ISO/IEC 25010 Quality Models⁹ to develop a comprehensive criteria catalog. The catalog realizes a qualitative evaluation since it does not include quantitative aspects like system load or performance. We then use this catalog to evaluate an example Saga pattern implementation using four different technological solutions. As many criteria of (Dürr et al., 2021) are adopted, refined but also extended, their analyzed solutions Eventuate Tram and Netflix Conductor are re-evaluated. While Eventuate Tram is a framework specifically designed for the Saga pattern, Netflix Conductor represents a workflow orchestration engine designed for distributed workflows in general. As a third solution, we consider MicroProfile LRA (backed by the Eclipse Foundation) since it represents a very recent solution explicitly designed for LRA in microservices, so also Sagas. The fourth and last evaluated solution is the workflow engine Camunda¹⁰ that can execute processes which are defined using *Business Process Modeling and Notation (BPMN)*. Since the BPMN language itself provides the option of defining compensation events for tasks, Rucker (Rucker, 2021) suggested using a BPMN-based workflow engine like Camunda to realize the Saga pattern. In addition, BPMN is an established standard, and Camunda was not considered in similar evaluations yet.

For Eventuate Tram (v0.14.0), Netflix Conductor (v2.30.3) and Camunda (v7.15.0), we implemented the sample application's microservices as Spring¹¹ services. In contrast, our MicroProfile LRA (v1.0) realization, implemented those using the MicroProfile runtime provided by OpenLiberty¹². More information concerning the realization can be found online¹³.

5 RESULTS

To perform a structured evaluation of different technological solutions for implementing the Saga pattern, clearly described and relevant criteria are needed. In the following we shortly describe the criteria catalog that we have developed and the evaluation of the technologies based on this catalog.

⁹<https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>

¹⁰<https://camunda.com/>

¹¹<https://spring.io/>

¹²<https://openliberty.io/docs/21.0.0.6/microprofile.html>

¹³<https://github.com/KarolinDuerr/BA-SagaPattern>

5.1 Criteria Catalog

Our criteria catalog considers several areas of interest related to characteristics of microservices and the Saga pattern as well as some quality attributes of the ISO/IEC 25010 standard. The focus is on how the Saga pattern can be implemented with the respective technologies and what they offer in addition for operating and maintaining a system based on the Saga pattern in production. We have therefore adopted and in some cases refined the criteria for **general Saga characteristics, monitoring, expandability, and fault tolerance** from (Dürr et al., 2021). In addition, organized by the quality attributes of the ISO/IEC 25010 standard, we have derived criteria considering the areas of **security, testability, and portability**. And finally, we adopted some **open-source characteristics** from (Confino and Laplante, 2010), because they can also be relevant aspects to consider when selecting a technology. Although some characteristics have to be treated with caution as they may be subject to different interpretations, like repository stars, or since no generally accepted metrics exist, for example regarding documentation (Confino and Laplante, 2010). We described all criteria in more detail in order to have a comprehensive and understandable criteria catalog that could also be applied to additional technologies. However, we cannot report the complete catalog due to space limitations here, but provide it online¹⁴. Furthermore, the different criteria are also presented in the following technological evaluation.

5.2 Technological Evaluation

In the following, we use all criteria from our catalog to evaluate the implementations¹⁵ based on the four technologies: the Eventuate Tram Saga framework, Netflix Conductor, Camunda, and MicroProfile LRA. Table 1 and table 2 summarize our results, of which some originate from the previous work by (Dürr et al., 2021) on Eventuate Tram and Netflix Conductor.

Considering **general Saga characteristics**, Conductor and Camunda offer a bit more flexibility, because Sagas can be *defined* in a language-independent way and for the *communication* between participants different options such as HTTP requests or Kafka¹⁶ messages are available. Instead, Eventuate provides a Java-based DSL and MicroProfile LRA requires Java Annotations to define Sagas based on JAX-RS resources. The communication between

participants is prescribed (Kafka messages for Eventuate and HTTP requests for MicroProfile LRA). Only Eventuate and Camunda allow to directly *link* the *compensating transactions* to transactions, which means only executed ones will be compensated (Dürr et al., 2021). With MicroProfile LRA, a corresponding compensating transaction needs to be defined in the same Java class to achieve this effect. Whereas for Conductor, per workflow, which represents a Saga as a collection of tasks, only one failure workflow can be specified which includes all compensating transactions. In case of an abort, the failure workflow is started and thus also unnecessary compensating transactions might be executed (Dürr et al., 2021). Conductor and Camunda rely on a central engine to *orchestrate* the Saga execution which in the case of Camunda can also be integrated with a service, e.g. the *TravelService*. The *TravelService* only starts a Saga execution then. With Eventuate and MicroProfile, the *TravelService* has the role of the orchestrator, but it relies on Eventuate's CDC service and the MicroProfile Coordinator respectively. Furthermore, Eventuate limits the Saga process execution to a strictly sequential one (Dürr et al., 2021) whereas the others allow for *parallel execution* of specified transactions. In the case of MicroProfile LRA, this is possible because it is the developer's responsibility to implement the Saga sequence. *Triggering* the *compensation process* for a Saga *externally* is possible with all technologies, except Eventuate, using the provided API. Though with Eventuate Tram participants can be connected directly (Dürr et al., 2021) enabling also an event-based choreography approach.

Our results for **monitoring** aspects partly stem from the previous evaluation by (Dürr et al., 2021). To retrieve *runtime information*, Conductor and Camunda provide a UI with several functionalities like visualizing current workflows. In contrast, with Eventuate all transactions and messages can only be accessed directly via the database, which could however be used to realize a custom visualization tool. Also MicroProfile LRA only provides a small set of runtime insights (LRA status: active/failed/closed) via the coordinator API. *Metrics* like average execution times or the number of failed Sagas can be collected from an endpoint for all technologies. Additionally, Eventuate Tram and MicroProfile LRA allow for easy activation of *distributed tracing*, using for example Zipkin or Jaeger¹⁷. Furthermore, all technologies provide sufficient *logging*. For the **expandability** criteria subset, we performed an extension of the implementations and results are again derived from (Dürr et al., 2021) where suitable. Expandability is facili-

¹⁴<https://karolinduerr.github.io/BA-SagaPattern/CriteriaCatalog/>

¹⁵<https://github.com/KarolinDuerr/BA-SagaPattern>

¹⁶<https://kafka.apache.org/>

¹⁷<https://www.jaegertracing.io/>

Table 1: Evaluation overview – Part 1.

Criterion	Eventuate Tram	Netflix Conductor	Camunda	MicroProfile LRA
General Saga Characteristics				
Saga definition	Eventuate DSL	JSON / provided clients	Modeler/ XML / Java DSL	Annotations
Orchestrating services	CDC Service, TravelService	Conductor server	Camunda Engine in TravelService	MicroProfile Coordinator, TravelService
Specifying compensating transactions	✓	✓	✓	✓
Compensation transaction allocation	specific transaction	entire workflow	specific transaction	JAX-RS resource class
Parallel transaction execution	✗	✓	✓	developer's responsibility
Parallel execution configurable	✗	✓	✓	developer's responsibility
Participant communication selectable	✗	✓	✓	✗
External compensation trigger	not directly	via API	via API	via API
Choreographed Sagas	✓	✗	✗	✗
Monitoring				
Runtime state of Sagas	via database	UI visualization	UI visualization, database	partly via coordinator API
Orchestrator metrics	from CDC service	from Conductor server	from embedded Process Engine	from embedded Coordinator, TravelService
Tracing	Zipkin integration	not directly supported	not directly supported	Zipkin & Jaeger integration
Logging	microservices logs	Conductor server logs	microservices logs	microservices logs
Expandability				
Terminating/Pausing running Sagas	not directly	via API / UI	via API / UI	via API
Versioning Sagas	✗	✓	✓	✗
Built-in language support	Java	Java, Python	Java	Java
Any language for orchestrator	✗	✓	✓	✗
Any language for participant	✓	✓	✓	✗
Fault Tolerance				
Execution timeouts	✗	enforced	enforced	✓
Reaction to participant fault	unsubscribe	retry	retry	developer's responsibility & compensation
Saga continuation trigger after orchestrator crash	new Saga instance	restart of Conductor server	restart of TravelService	✗ (developer's responsibility)
New Sagas while orchestrator unavailable	✓	only with buffering	✗	✗
Independent compensating transactions	✓	✗	✓	✓
Orchestrator reaction to duplicate messages	detect & ignore	detect & ignore	log exception & ignore	developer's responsibility, detect & ignore
Orchestrator reaction to old messages	detect & ignore	detect & ignore	log exception & ignore	developer's responsibility, detect & ignore
High availability	through replication	through replication	through replication	–

Table 2: Evaluation overview – Part 2.

Criterion	Eventuate Tram	Netflix Conductor	Camunda	MicroProfile LRA
Security				
Encrypted Communication*	✓	✓	✓	✓
Authentication Support*	✗	✗	✓	MicroProfile JWT
Authorization Support*	✗	✗	✓	MicroProfile JWT
Testability				
In-house test framework	✓	✗	✓	✗
Unit test Saga definition	✓	✗	✓	successful Saga
Unit test Saga participant	✓	✓	✓	✓
Saga integration test*	✓	✓	✓	✓
Portability				
Containerization	Docker Hub image	Dockerfile & community Docker Hub image	Docker Hub image	Docker Hub image**
	Kubernetes examples	✗	Kubernetes examples	Kubernetes examples**
Cloud Provider	✗	✗	Camunda Cloud	several guides**
OSS Characteristics				
Provider	Eventuate	Netflix	Camunda	Eclipse
License Type	Apache Licence 2.0	Apache Licence 2.0	Apache Licence 2.0	Apache Licence 2.0
Repository Stars***	781	3664	2350	71
Fork Count***	180	approx. 1200	approx. 1000	24
Contributor Count***	2	161	175	18
Support	Issue tracking Discussion forum Enterprise support	Issue tracking Discussion forum	Issue tracking Discussion forum Enterprise support	Issue tracking Discussion forum
Documentation	✓	✓	✓	✓

* Not implemented by the prototype ** Depends on chosen MicroProfile runtime, here: OpenLiberty

*** Numbers specific for respective Github repository, as of 2021-11-30

tated the most by Conductor and Camunda, as both allow to *pause or terminate currently running Sagas* either via their API or UI and they support *versioning* Sagas so that Sagas of a new version can start while a previous version still runs. Also, considering the usage of different *programming languages*, Conductor and Camunda do allow both the orchestrator and the participants to be written in almost any language, because the definition of Sagas is language-independent. MicroProfile LRA is restricted, because the MicroProfile specification is explicitly for Java-based microservices, while Eventuate prescribes Java only for the orchestrator so that participants could be written in any language. This is possible, although *built-in support* is only provided for Java, which is similar to the others, except for Conductor which also offers a Python client.

Regarding aspects of **fault tolerance** to handle distributed systems challenges, we found that Conductor, Camunda, and Eventuate retry communica-

tions in case of *participant crashes*. However, Eventuate unsubscribes participants responding with a failure and they are re-registered only after a restart (Dürr et al., 2021). MicroProfile LRA itself provides no handling, but it can be combined with MicroProfile Fault Tolerance¹⁸. *Execution timeouts* are enforced by Conductor and Camunda and can be defined with MicroProfile LRA, so that compensation is triggered if the timeout is reached, whereas Eventuate might wait indefinitely (Dürr et al., 2021). All technologies, except Conductor, can tolerate failures during compensation for participants where no *compensation* is necessary, because of their *independent* specification. *Orchestrator crashes* are tolerated by persisting the Saga state, only MicroProfile LRA leaves this to the developer. However, Eventuate needs an external trigger like a new Saga start to continue (Dürr et al.,

¹⁸<https://github.com/eclipse/microprofile-fault-tolerance>

2021), while Conductor and Camunda continue after a restart. Starting *new Sagas while the orchestrator is unavailable* is only possible when a separate service is responsible for the Saga start as in the case of Eventuate (Dürr et al., 2021) or when the orchestrator would be a service separate from the *TravelService*. Otherwise, buffering requests is another option. All technologies, except MicroProfile LRA, detect and ignore **duplicated** or *old messages* by default. Finally, all technologies can be set up to be *highly available* by replicating their respective orchestrators, except for MicroProfile LRA where the chosen runtime, OpenLiberty, does not include explicit information on replication. Regarding **security** criteria, all technologies support *encrypted communication*, although partly not built-in. Extensive *authentication and authorization support*, however, is only provided by Camunda for example by defining user groups. MicroProfile LRA can be combined with MicroProfile JWT Auth¹⁹ to include authentication and authorization. Because proper Saga implementations should also include tests (Richardson, 2019), the technologies should ideally facilitate **testability**. Of the four evaluated technologies, only Eventuate and Camunda provide *in-house test frameworks* which also allow testing the *Saga definition in isolation*. With MicroProfile LRA, currently only the success case of a Saga execution can be tested due to the annotation-based compensation mechanism. Nevertheless, testing *Saga participants in isolation* and writing *integration tests* is possible with all four technologies. To support **portability**, technologies should be deployable on different platforms. All support *containerization* via Docker and Kubernetes, although for Conductor only a community image is available and no examples for Kubernetes are given. In the case of MicroProfile LRA, deployability depends on the chosen runtime, in our case OpenLiberty. Deployment instructions for specific *cloud providers* are merely available for Camunda with Camunda Cloud²⁰ and MicroProfile LRA which provides several guides²¹. Finally, considering **OSS characteristics**, it can be stated that Conductor and Camunda show a higher *community interaction*, which might be due to the popularity of the providers behind them. Instead, Eventuate has a very specific focus and MicroProfile LRA is very recent (its first release was in May 2021). *Support options* are equally well, only Eventuate and Camunda also offer a paid enterprise version with additional support. Again, for MicroProfile LRA, the support depends on the chosen runtime. Each technology also maintains detailed and

clear *documentation* including examples. In terms of *license type*, all solutions have opted for the Apache Licence 2.0 which ranks among the most permissive licenses (Confino and Laplante, 2010).

6 DISCUSSION AND FUTURE WORK

With our criteria catalog in Section 5.1 we provide an answer to **RQ1**. It covers a comprehensive set of criteria which can be used to evaluate technological solutions for implementing the Saga pattern. The focus is on how well an implementation of the Saga pattern is supported, also considering operational aspects, like monitoring or expandability. Except for the criteria *Saga definition* and *Specifying compensating transactions* which are basic requirements, the importance of the criteria depends on the specific context in which a system is implemented and no general order of importance for the criteria can be established. For example, *Authentication Support* might not be that important in a context where all services run in an isolated network while *Containerization* is important for flexibility regarding the execution environment.

To use the catalog for an evaluation of a technology, in principle, the documentation of a technology is sufficient if it provides detailed enough information. But to have reliable results, an actual implementation using the technology is needed, especially considering the criteria for *Fault Tolerance*. Our answer to **RQ2** is therefore based on the implementations and presented in Section 5.2. While, in summary, all four evaluated technologies allow robust implementations of the Saga pattern, we identified the following differences. Since Netflix Conductor is not specifically designed for Sagas, it does not represent the respective characteristics as clearly as the other frameworks. However, it allows for more configuration options like the chosen communication mechanism. Camunda is somehow similar but represents the Saga characteristics better due to the usage of BPMN which provides suitable entities (e.g., service tasks or compensation events) to design Sagas. While Eventuate Tram and MicroProfile LRA are actually designed for the Saga pattern, they also come with more limitations concerning configurability. Nevertheless, a choreographed approach is only supported by Eventuate Tram. Good Saga testability support is solely provided by Camunda and Eventuate Tram, as they allow running tests in isolation. When extending an existing system, Netflix Conductor and Camunda allow for greater flexibility, among other things, through the possibility of versioning. Additionally, Conductor's

¹⁹<https://github.com/eclipse/microprofile-jwt-auth>

²⁰<https://camunda.com/products/cloud/>

²¹https://openliberty.io/guides/#cloud_deployment

and Camunda's UI are beneficial tools when monitoring a system. On the other hand, the ability to activate tracing without additional implementation effort is only provided by Eventuate Tram and MicroProfile LRA. All four technologies offer some portability support. However, only Camunda and MicroProfile LRA (depending on the runtime) provide specific cloud provider support. Security-related features are currently only available with Camunda. For MicroProfile LRA, additional MicroProfile extensions could be used to realize security aspects. All technologies consider fault tolerance, also with some configuration options. Only MicroProfile LRA leaves some aspects to the developer's responsibility. Concerning OSS characteristics, documentation and support exist for all technologies.

Because the importance of the considered criteria differs depending on the context and the evaluated solutions differ in how they support the criteria, it is not possible to make a statement about whether one solution is better than another. Nevertheless, our evaluation can be used to make an informed decision on which solution fits which context. In future work, additional solutions can be evaluated or the catalog can be extended with quantitative measures, for example considering performance or resource utilisation.

7 CONCLUSION

Because implementing the Saga pattern involves considerable complexity, technologies supporting it have emerged from which a suitable one can be chosen. To make such a choice in an informed way, our work presents a criteria catalog for evaluating Saga pattern implementation technologies and applies it to four existing solutions. Based on our findings, the considered technologies differ according to how they support the criteria and no technology is superior to the others. Our evaluation can therefore be used to select a suitable technology for a specific context in which the Saga pattern should be implemented.

REFERENCES

- Al-Houmailya, Y. J. and Samaras, G. (2009). Two-Phase Commit. In *Encyclopedia of Database Systems*, pages 3204–3209. Springer US.
- Alshuqayran, N., Ali, N., and Evans, R. (2016). A Systematic Mapping Study in Microservice Architecture. In *9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51. IEEE.
- Bruce, M. and Pereira, P. A. (2018). *Microservices in Action*. Manning Publ., 1st edition.
- Confino, J. P. and Laplante, P. A. (2010). An Open Source Software Evaluation Model. *Int. J. Strateg. Inf. Technol. Appl.*, 1(1):60–77.
- Cruz, D., Wieland, T., and Ziegler, A. (2006). Evaluation Criteria for Free/Open Source Software Products Based on Project Analysis. *Software Process: Improvement and Practice*, 11(2):107–122.
- Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazza, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer.
- Dürr, K., Lichtenthäler, R., and Wirtz, G. (2021). An Evaluation of Saga Pattern Implementation Technologies. In *13th European Workshop on Services and their Composition (ZEUS)*, volume 2839, pages 74–82. CEUR-WS.org.
- Estdale, J. and Georgiadou, E. (2018). Applying the ISO/IEC 25010 Quality Models to Software Product. In *Systems, Software and Services Process Improvement - 25th European Conference (EuroSPI)*, volume 896 of *CCIS*, pages 492–503. Springer.
- Fugaro, L. and Vocale, M. (2019). *Hands-On Cloud-Native Microservices With Jakarta EE*. Packt Publ., 1st edition.
- Gaffney, J. E. (1981). Metrics in Software Quality Assurance. In *Proceedings of the ACM 1981 Annual Conference*, pages 126–130. ACM.
- Garcia-Molina, H. and Salem, K. (1987). Sagas. In *Proceedings of the 1987 SIGMOD International Conference on Management of Data*, volume 16, pages 249–259. ACM Press.
- Helland, P. (2016). Life Beyond Distributed Transactions: An Apostate's Opinion. *ACM Queue*, 14(5):69–98.
- Limón, X., Guerra-Hernández, A., Sánchez-García, A. J., and Arriaga, J. C. P. (2018). SagaMAS: A Software Framework for Distributed Transactions in the Microservice Architecture. In *6th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 50–58. IEEE.
- Newman, S. (2015). *Building Microservices - Designing Fine-Grained Systems*. O'Reilly Media, 1st edition.
- Newman, S. (2019). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, 1st edition.
- Rücker, B. (2021). *Practical Process Automation*. O'Reilly Media, 1st edition.
- Richardson, C. (2019). *Microservices Patterns*. Manning Publ., 1st edition.
- Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. (2012). Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD International Conference on Management of Data*, pages 1–12. ACM.
- Vossen, G. (2009). ACID Properties. In *Encyclopedia of Database Systems*, pages 19–21. Springer US.
- Štefanko, M., Chaloupka, O., and Rossi, B. (2019). The Saga Pattern in a Reactive Microservices Environment. In *Proceedings of the 14th International Conference on Software Technologies (ICSOT)*, pages 483–490. SciTePress.