

ASO-Saga: Adaptive Saga Orchestration with Parallel Execution and Batched Compensation for Distributed Transactions in Enterprise Microservices

Amit Mishra

Senior Software Engineer Distributed Systems and Cloud Computing

ARTICLE INFO

Received: 01 Nov 2024

Revised: 18 Dec 2024

Accepted: 29 Dec 2024

ABSTRACT

In a microservice architecture, the Saga pattern is the most widely used way to achieve consistency between databases. In standard sagas, steps are always completed one after the other, coordinated in the same way and rolled back one at a time in case of a failure. However, each transaction is slow, and recovery time is proportional to the number of services. In this paper, we propose ASO-Saga, an adaptive saga orchestration framework with three mechanisms. First, independent steps are executed concurrently at the same part of the dependency graph, and second, the hybrid dispatcher executes compensable steps in a command and final non-compensable steps in an event. Third, batched idempotent compensation cancels all completed steps in parallel using a small saga log. A multi-process prototype using one database per service was compared to sequential orchestration and event choreography in an e-commerce order saga. With 1 to 25 virtual users, an increase of 15% to 32% in throughput and a decrease of 18% to 27% in median latency was achieved with ASO-Saga. The baseline recovery time went from 3.5 ms to 23.5 ms with 2 to 8 services. ASO-Saga also kept the recovery time below 4.1 ms (83% faster). When the message loss was between 0.5 and 2%, ASO-Saga and orchestration kept every failed saga consistent, while choreography failed in 9% to 27% of cases. An ablation study shows that two optimizations from ASO-Saga are parallel execution and batched compensation. One trade-off is that failure at one level in parallel execution requires an additional compensation for the sibling step. This thus leads to improvements in saga performance while preserving orchestration consistency guarantees.

Keywords: Saga pattern; enterprise microservices; distributed transactions; service orchestration; event-driven architecture; compensating transactions; fault tolerance; data consistency; performance optimisation

1. INTRODUCTION

Enterprise applications are increasingly using microservices, where each microservice has its database, and the services are individually deployed, scaled, and upgraded (Jamshidi et al., 2018). This improves agility and scalability but sacrifices some advantages of a single database, such as transactional integrity. For operations, such as when an order is placed, a payment is settled, or a shipping is booked, multiple services and data stores are involved, and therefore the required consistency can only be provided by a coordination protocol, rather than one local ACID transaction (Laigner et al., 2021). Distributed data management is reported by practitioners to be among the most challenging aspects of microservice implementations (Soldani et al., 2018; Zhou et al., 2021) and failures that cross service boundaries are hard to debug.

Two families of solutions exist. Atomic commit protocols such as two-phase commit offer strong isolation. They rely on locks across service boundaries, which limits availability and throughput in loosely coupled systems (Fan et al., 2020; Zhang et al., 2019). The Saga pattern takes a different approach. It breaks a distributed transaction into a sequence of local transactions, with a compensating action attached to each transaction to undo its effect if a later transaction fails (Štefanko et al., 2019). Sagas trade isolation for availability. They are now the default mechanism

for consistency for enterprise microservice frameworks, either via centralized orchestrator or choreographed events (Rudrabhatla, 2018).

Also, by definition, the base Saga pattern takes every step linearly, even when there are no dependencies among them. This means that end-to-end latency is equal to the sum of the service times of every step. The same style of coordination is used in all steps, though orchestration and choreographies differ considerably in terms of cost and reliability (Rudrabhatla, 2018). During failure recovery, compensating actions are re-executed one by one in reverse order, resulting in a longer recovery time if more steps are present. Related work on failure recovery has focused on the missing isolation property (Daraghmi et al., 2022) or tool support for sagas (Limón et al., 2018; Štefanko et al., 2019). Other commit protocols that can replace sagas have also been proposed (Fan et al., 2020; Kraft et al., 2023), but few studies investigated the performance or robustness of sagas under system load.

This paper addresses that gap. The proposed framework, called ASO-Saga (Adaptive Saga Orchestration), preserves the semantics of the classical saga pattern. It changes only the way these steps are scheduled, dispatched and paid for. Its main contributions are:

- Dependency-aware execution. Since a saga can be treated as a directed acyclic graph, independent steps can be executed in parallel as levels. A bound on the resulting latency is given.
- A hybrid coordination policy where compensable steps are sent as orchestrated command and reply pairs, and non-compensable final steps are sent as choreographed events.
- A batched idempotent compensation mechanism, where all completed steps are undone in parallel using a compact saga log, so as to prevent recovery time from growing with saga length.
- We built a multi-process prototype, where each service has its own database, and compared orchestration and choreography by measuring correctness, throughput, and latency Under Load, response time Under Error Injection, recovery time based on the saga's length, consistency Under Message Loss, and performing an ablation study.

2. LITERATURE REVIEW

Microservices have been the subject of systematic literature reviews that have identified both advantages and disadvantages of microservice architecture (Di Francesco et al. 2019). For example, they found data management and inter-service communication had the least mature solutions among all microservices characteristics. Waseem et al. (2020) come to a similar conclusion in the context of microservices in a DevOps setting, citing a lack of support for inter-service consistency. A grey literature review by Soldani et al. (2018) shows that practitioners see distributed transactions as one of the main pains of the style, and performance studies show that this comes as a communication overhead. Blinowski et al. (2022) show that the latency of inter-service calls dominates the latency when comparing an environment with N microservices versus an equivalent monolith, as N grows. This means that there is a need to find ways to reduce the coordination cost of cross-service interactions instead of treating them as a fixed price of modularity.

Research on distributed transactions in this context has gone into two directions, one of which maintains strong consistency. GRIT achieves deterministic execution and optimistic concurrency control to provide serialisable transactions for polyglot microservices (Zhang et al., 2019). 2PC* replaces synchronous locking with an asynchronous optimistic lock to increase concurrency, and has been evaluated on a middleware prototype implemented on top of Spring Cloud (Fan et al., 2020). Another extension of this line to mixed data stores is Epoxy, which adds multi-version concurrency control to existing databases (Kraft et al., 2023). These protocols guarantee isolation. However, they do require the presence of a shared transaction manager or integration at the database level, which goes against the loose-coupling which microservices are expected to provide (Laigner et al., 2021).

The second direction is eventually consistent applications using the Saga pattern. One example in Štefanko et al. (2019) studies the usage of the Saga pattern in reactive systems with four different Java frameworks. They found large differences in implementation complexity and throughput. Limón et al. (2018) proposed SagaMAS, a multi-agent layer that coordinates sagas on behalf of services. Rudrabhatla (2018) built orchestration and choreography atop Spring Boot, MongoDB and ActiveMQ. Measuring the rollback times for growing numbers of services from two to eight, he found choreography to be faster for small sagas but orchestration more understandable and easier to manage as the number of services grows. Daraghmi et al. (2022) built a five-service e-commerce prototype using Spring Boot, Kafka, Redis and PostgreSQL, and validated both the success and failure scenarios based on the service logs. They measured throughput with up to 150 virtual users and the response times under error rates of 10%, 40% and 70%. They add the isolation property by storing the intermediate writes in a quota cache. Event sourced systems often use persistent logs in practice for reconstructing the state of a saga and for recovery from failures (Overeem et al., 2021). For instance, fault studies show that the improper handling of partial failures across services causes many production failures in microservice systems (Zhou et al., 2021).

Benchmarking practice shapes claims made about such mechanisms. Gan et al. (2019) released DeathStarBench, which provides multi-step transactional workflows for benchmarking studies. They also showed that the tail latency in a microservice system is determined by the longest dependency chain, which motivates our work on shortening the critical path of a saga. Table 1 compares the studies to this study.

Collectively, these studies establish the Saga pattern as the practical consistency mechanism for enterprise microservices and establish evaluation practices that it argues involve prototypes, load tools, and error injection. However, each of these studies addresses only one aspect. Isolation add-on and agent-based coordination, as well as the framework comparison, do not affect the sequential model, the single coordination style, or the serial recovery of the classical pattern. However, they do not investigate the joint combination of step scheduling, coordination strategy, and compensation strategy, or evaluate them on the same real system under load, error-injection and message-loss. ASO-Saga fills this gap.

Table 1. Comparison of representative studies on distributed transactions in microservices with the present work.

Study	Consistency model	Parallel step execution	Hybrid coordination	Parallel / batched compensation	Evaluation
Zhang et al. (2019) – GRIT	Serialisable	No	No	N/A (abort)	Prototype demo
Fan et al. (2020) – 2PC*	Strong (locks)	No	No	Sequential	Middleware prototype, TPS vs users
Limón et al. (2018) – SagaMAS	Eventual (saga)	No	No	Sequential	Framework design

Štefanko et al. (2019)	Eventual (saga)	No	No	Sequential	Four frameworks, one scenario
Rudrabhatla (2018)	Eventual (saga)	No	Compared, not combined	Sequential	Prototype, rollback vs 2–8 services
Daraghmi et al. (2022)	Eventual + isolation cache	No	Both used, static	Cache-level	Prototype, 1–150 VUs, 10–70% error
Kraft et al. (2023) – Epoxy	Snapshot isolation	No	No	N/A (abort)	Prototype, benchmarks
This work – ASO-Saga	Eventual (saga)	Yes (DAG levels)	Yes (per step)	Yes (parallel, idempotent)	Prototype, 1–150 VUs, 10–70% error, 2–8 services, message loss, ablation

3. METHODOLOGY

3.1 System architecture

Figure 1 shows the architecture of our proposed framework. The API gateway receives a request from a client and then starts a saga with the ASO-Saga coordinator. The saga definition consists of the step graph including compensating action for each step. The level scheduler transforms this graph into execution levels where steps on the same level are free of dependencies. The hybrid dispatcher then sends a step either as a command with synchronous reply or as an event to the message broker. The saga log is small as it only contains one log entry per state change. The coordinator can recover and a batched compensator uses the log to send idempotent compensating commands to all services that have been previously completed when a step fails. The outcome monitor persists step results and timings. Each service maintains a local store and publishes a forward action, compensating action and idempotency-key table that allows the framework to behave correctly without modifying the internals of each service.

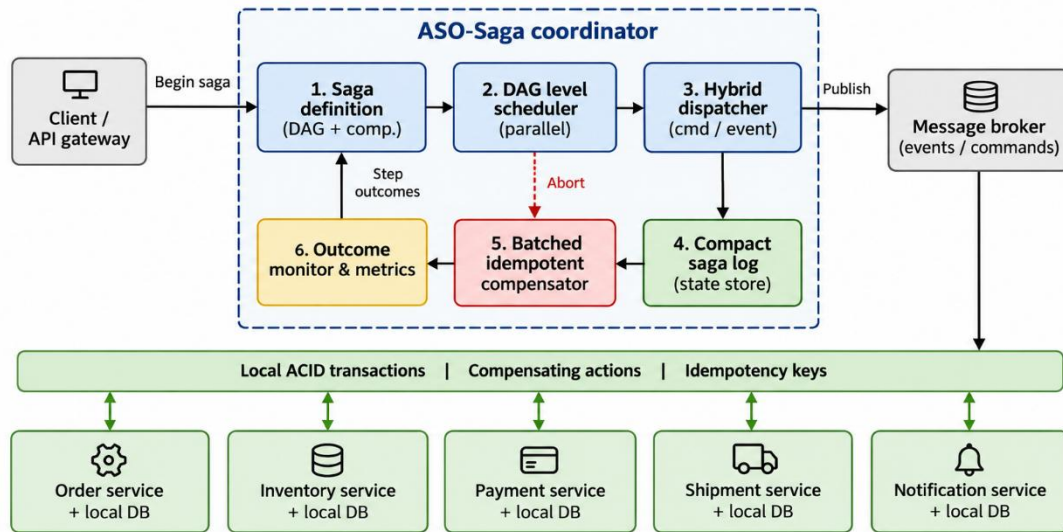


Figure 1. Architecture and workflow of the proposed ASO-Saga framework. Numbered components form the coordinator; the dotted arrow denotes the abort path.

3.2 Saga model and level-parallel execution

We define a saga to be a directed acyclic graph (DAG) $G = (V, E)$ where $V = \{T_1, \dots, T_n\}$ is the set of local transactions and (T_i, T_j) is an edge if T_j reads data written by T_i . Each step T_i has a compensation action C_i , an expected service time s_i , a compensation time c_i , and a coordination delay d_i (the number of hops and number of writes the step makes). The scheduler partitions V into levels $L_1, \dots, L_k$ according to the longest-path depth, placing all the predecessors of steps in earlier levels. In a successful saga where all steps of a level run in parallel, the end-to-end latency under sequential and level-parallel execution is:

$$\Lambda_{seq} = \sum_{i=1}^n (s_i + d_i), \quad \Lambda_{par} = \sum_{k=1}^K \max_{T_i \in L_k} (s_i + d_i) \quad (1)$$

Since $K \leq n$ and each level contributes only its slowest step, $\Lambda_{par} \leq \Lambda_{seq}$ for any graph, with equality only for a chain. In the order saga used in this study, inventory reservation and payment authorisation depend only on order creation. They can share a level, so the shorter of the two leaves the critical path.

3.3 Hybrid coordination

Each step of an orchestrated command requires two hops with a message and a write to the log. The coordinator receives an explicit reply. This simplifies failure checking and retry. One hop and one broker publish are also consumed by a choreographed event. Thus, a lost or duplicate event puts the saga in an unknown state because no one is waiting for a reply (Rudrabhatla, 2018). ASO-Saga executes one rule at a time. If the step has any compensating action, it is sent as a command. If the last step has no compensating action, such as a customer notification, it is sent as an event. This permits every step necessary for recovery to be controlled explicitly, thereby eliminating a round trip and a log write for every step whose failure would never need compensating. Each log record contains only the saga identifier, step identifier, outcome, and the selected idempotency key, which lowers the persistence costs.

3.4 Batched idempotent compensation

In a standard backward recovery, compensations are executed sequentially in reverse order. In ASO-Saga, a compensation is idempotent and identified by a saga identifier that is passed as a key. They can therefore run in any order and can be safely retried. When one of them fails, the compensator reads the set D of completed steps from the saga log and sends one compensating command per service. Recovery time becomes limited by the slowest form of compensation rather than the sum.

$$R_{seq} = \sum_{j \in D} (c_j + d_j), \quad R_{batch} = \max_{j \in D} (c_j + d_j) + \delta \quad (2)$$

where δ is the cost of one log read. A service that has already applied a given key returns success without repeating the action. Retries after a lost message therefore do not corrupt state. Parallel execution brings one trade-off. When step T_f in level L_k fails, its siblings in the same level may already be complete and must also be compensated. A sequential saga would never have started them. The expected extra compensation cost of a failure in level k is:

$$E[\Delta C_k] = \sum_{T_j \in L_k, j \neq f} \Pr(T_j \text{ completes before } T_f \text{ fails}) \cdot c_j \quad (3)$$

This is the cost incurred on any failure within a multi-step level and is bounded by the compensation time of any siblings. The saving in latency observed in Equation (1) occurs for every saga that is completed successfully. The complete execution and recovery process is detailed in Algorithm 1.

Algorithm 1. ASO-Saga execution and recovery

Input: saga graph $G = (V, E)$ with compensating actions; payload

Output: COMMITTED or COMPENSATED

```

1: levels  $\leftarrow$  topological levels of  $G$  (Section 3.2); done  $\leftarrow \emptyset$ 
2: for each level  $L_k$  in order do
3:   log(saga,  $L_k$ , STARTED)
4:   for each  $T_i$  in  $L_k$  do           // issued together, no waiting between steps
5:     if  $T_i$  has no compensating action then publish event( $T_i$ )
6:     else send command( $T_i$ ) with idempotency key (saga,  $i$ )
7:   await all replies; re-send any command whose reply timed out
8:   if any  $T_i$  failed then
9:      $D \leftarrow$  done  $\cup$  {completed compensable siblings in  $L_k$ }
10:    send compensate(saga,  $j$ ) to all  $j \in D$  at once; await replies; re-send on timeout
11:    log(saga, COMPENSATED); return COMPENSATED
12:   done  $\leftarrow$  done  $\cup$  {compensable  $T_i$  in  $L_k$ }; log(saga,  $L_k$ , DONE)
13: return COMMITTED

```

3.5 Prototype implementation

The baselines and the framework were implemented in a Python version 3.12 prototype with e-commerce architectures inspired by Daraghmi et al. (2022) and Rudrabhatla (2018) using five services (Order, Inventory, Payment, Shipment, and Notification) that are run in five separate operating-system processes. Each of them uses its own write-ahead-logging SQLite database, with 64 worker threads in a thread pool. It maintains a single table of records, which stores the business state of each saga, and a table of applied keys that make its forward and compensating actions idempotent. A separate broker process handles topic-based publish/subscribe of events. Commands and events are sent over operating-system pipes. Each command is sent by the coordinator with a 250 ms timeout and up to six retries, made safe with idempotency keys. Business logic is emulated with a fixed wait time per step (including downstream calls to an external payment gateway or a carrier API) and a real database transaction. The numbers used in the experiments can be seen in Table 2; in the sequential orchestration version (following the reference design), the central orchestrator sends commands and waits for a response before sending the next command. If a command fails, it rolls back in reverse order. In event choreography modeled after the reference design, each service publishes the next event after its local transaction and in case of failure, a compensation event to its predecessor. The client waits for up to 3 s for a completion or compensation event and ASO-Saga runs the Algorithm 1 on the graph with an Inventory and Payment of the same level.

Table 2. Prototype and workload parameters of the order-processing saga.

Step (service)	Emulated work (ms)	Compensation (ms)	Depends on	Role in experiments
T1 Order	2.0	1.5	—	Always succeeds
T2 Inventory	4.0	3.0	T1	Injected failure (40% of failures)
T3 Payment	8.0	7.0	T1	Injected failure (60% of failures)
T4 Shipment	5.0	4.0	T2, T3	Always succeeds
T5 Notification	1.5	none	T4	Non-compensable; event in ASO-Saga
Environment	Ubuntu 24.04 container, single CPU core; 7 processes (5 services, broker, coordinator/load generator); SQLite 3 with WAL; 64 worker threads per service; command timeout 250 ms, 6 retries; 12 s per load run; message loss injected uniformly on every delivery when enabled.			

3.6 Experimental design and metrics

Six experiments were performed. In a similar fashion to the prototype studies described above, correctness was tested in experiment E1 by traversing a successful and a failed saga through the saga log and five databases. Throughput and latency are measured at E2 with 1, 5, 10, 25, 50, 100 and 150 closed-loop virtual users and 10% error. Response time is measured at E3 with 25 virtual users and 10%, 40% and 70% error rates. E4 is the backward recovery time for the last of 2, 4, 6 or 8 chained services that execute 4 ms normal work and 3 ms compensation, averaged over five runs of forty sagas. E5 is the evaluation of consistency with 25 virtual users in presence of lost messages at loss rate 0%, 0.5%, 1% and 2%, with error rate 40%. E6 assesses how much each of the mechanisms in ASO-Saga contributes to the system. It disables one mechanism at a time and tests with 25 virtual users under 40% error rate. Four measures are used. Throughput is the number of sagas successfully completed per second. Latency is the mean, median and 95th percentile of the time from a saga's start to its commit or complete compensation. The recovery time is the time from detection of failure until the last compensation is acknowledged. The consistency rate

is the proportion of failed sagas for which no service database still holds a committed forward record, and is determined using an audit of all service databases after each run. Due to the limitation of one CPU core shared between all processes, saturation throughput for all strategy implementations is 1300-1600 sagas per second. Results above 25 virtual users therefore indicate behavior at host saturation rather than a limit of the design itself.

4. RESULTS AND DISCUSSION

4.1 Validation of correct execution and recovery

Figure 2 reproduces the execution of one saga per scenario from the saga log. In the success scenario, orchestration executes its five steps sequentially in 22.2 ms total. ASO-Saga (subprocesses sent in one batch as Inventory + Payment) takes 2.4 ms, finishes the saga in 16.3 ms. Event sent for notification as part of saga process with no gap on the critical path. After the two cases, all five databases have a DONE record for the saga. In the case of payment failure, orchestration sees it at 14.9 ms and returns after compensating Inventory and Order sequentially at 20.2 ms. ASO-Saga sees it at 10.6 ms and returns after compensating Order and Inventory in parallel at 14.2 ms. In both cases, audit sees Order and Inventory COMPENSATED and no record of Payment, Shipment and Notification. This shows that the framework preserves the recovery semantics of the classic pattern while shortening both the forward and the backward path.

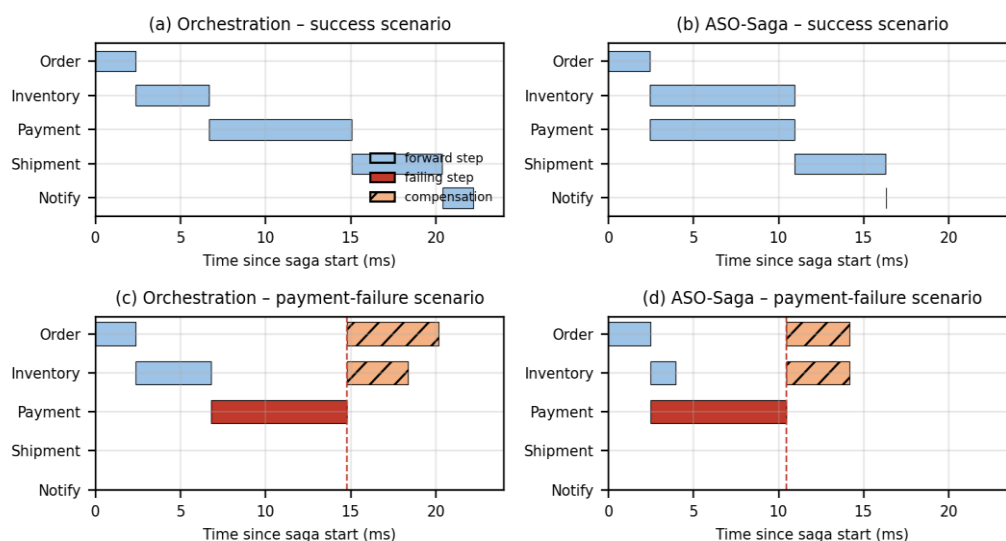


Figure 2. Execution traces reconstructed from the saga log for the success scenario (a, b) and the payment-failure scenario (c, d). Hatched bars are compensating actions; the dashed line marks failure detection.

4.2 Throughput and latency under load

Figure 3 shows throughput and latency for an increasing number of virtual users. ASO-Saga achieves 61 sagas per second for one user, whereas both baselines achieve 46, a 32% improvement. This leads to a median latency of 16.2 ms; compared to 22.2 ms, an improvement of 27%. This matches Equation (1) where the 4 ms inventory change was stripped off the critical path and an event was generated. The throughput with 10 users is 28% (or 26%) higher than orchestration (or choreography); with 25 users throughput is higher by 15%. The median latency is also lower, 20.9 ms vs 25.5 ms, or 25.0 ms. Over 50 users, a single-core host saturates, so we see similar throughput with all three approaches, 1300-1600 sagas/sec. The pattern starts winning at this point: Choreography consumes 5 messages per saga, Orchestration consumes 10 messages, and ASO-Saga consumes 9 messages. After the host is saturated, message handling is the dominant CPU consumer. The main advantage of ASO-Saga is the reduction of critical path and

coordinating latency, not CPU usage. For a multi-core setup, the saturation point would move outwards for all strategies and the advantage of lower latency would extend over a larger range.

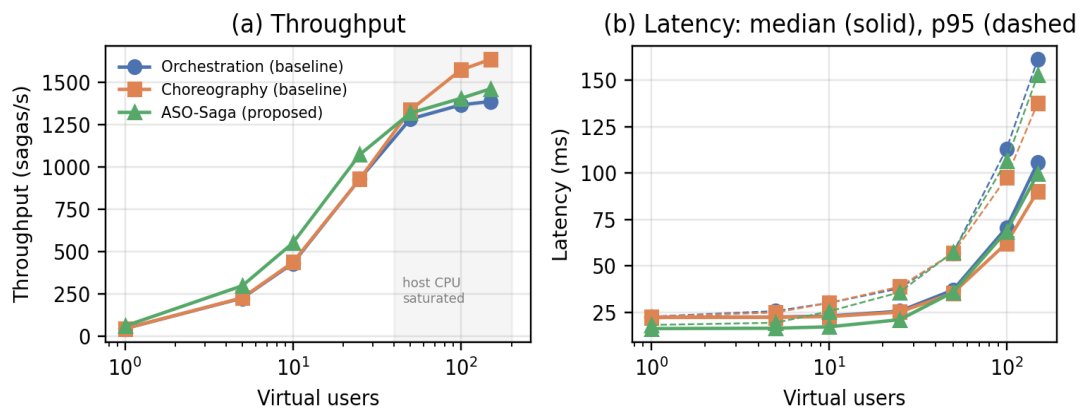


Figure 3. (a) Throughput and (b) latency versus number of virtual users at a 10% error rate. The shaded region marks saturation of the single-core host.

4.3 Response time under error injection

Figure 4 plots the response time as the error rate increases from 10% to 70% with 25 virtual users. ASO-Saga has the least mean response time and tail latency at all error rates. 10% error has 23.8 ms mean (16% faster than 28.3 ms baseline) and 38.1 ms 95th percentile (compared to 43.3 ms and 42.9). With higher error rates all have marginally faster means because failure ends sagas early. The three strategies differ 5% at 70% error. The recovery time is also different, with ASO-Saga taking 7.1-7.4ms to recover a failed saga, orchestration taking 5.4-6.1ms and choreography 5.7-6.3ms. While each injected failure (across the Inventory-Payment step) needs to be compensated by the sibling step that has already updated (and failed), the 7 ms payment compensation is the dominant and only factor that affects time. However, the total time for a failed saga is still lower for ASO-Saga because the forward path is relatively shorter (see Figure 2, the recovery path is longer).

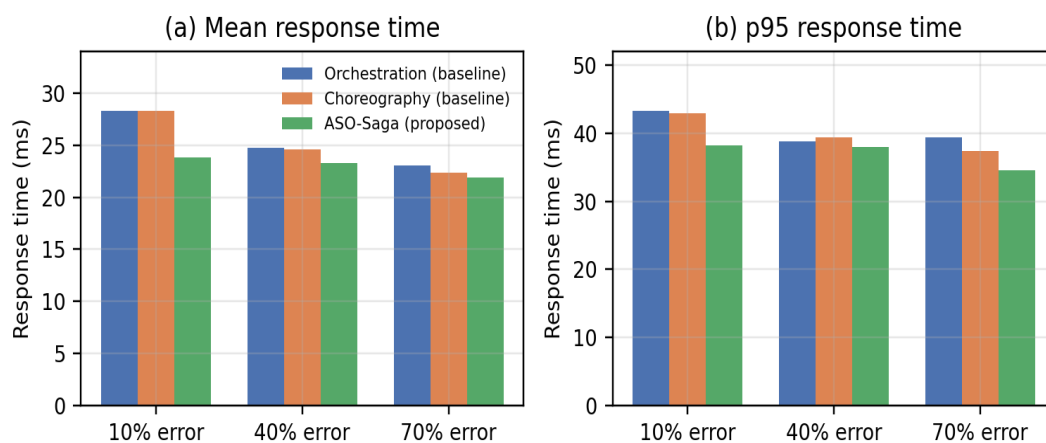


Figure 4. (a) Mean and (b) 95th-percentile response time with 25 virtual users at 10%, 40% and 70% error rates.

4.4 Recovery time versus number of participating services

In Figure 5, which reproduces an experiment by Rudrabhatla (2018) in which the last service fails for a chain of 2, 4, 6 or 8 services, the recovery time grows linearly with the number of completed steps in the baseline conditions: from 3.5 ms with two services to 23.5 ms with orchestration and 23.4 ms with choreography with eight services, because

ASO-Saga compensates all completed steps in one execution. Its recovery time (3.4 to 4.0 ms) is 83% lower at 8 services, with a standard deviation of under 0.13 ms. Because our chain uses no parallelism, this experiment isolates the impact of batched compensations. The end-to-end time of the failed saga is 33% lower for ASO-Saga at eight services (Figure 5(b)) because, as the number of services in the chain increases, recovery becomes a larger part of the time of the failed saga's execution path. The result shows where each mechanism pays off: parallel execution helps sagas with wide graphs, while batched compensation helps sagas with many steps, regardless of the shape of the graph.

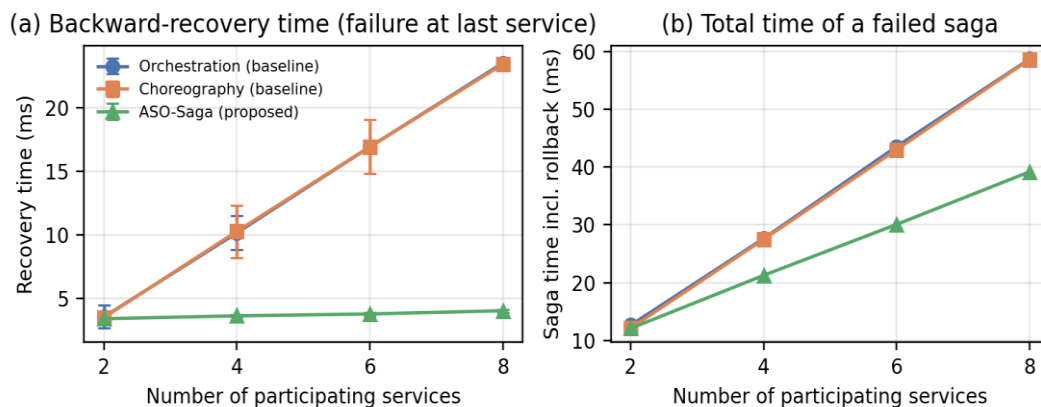


Figure 5. (a) Backward-recovery time and (b) total time of a failed saga as the number of chained services grows from 2 to 8 with failure at the last service (mean of five runs; error bars show one standard deviation).

4.5 Consistency under message loss

With 25 virtual users and a 40% error rate, the results for uniform message loss shown in Figure 6 indicate that all failed sagas are 100% consistent for orchestration and ASO-Saga for any message loss rate. All commands and compensations are retried on timeout. Since idempotency keys make retry safe, choreography provides 91.1% at 0.5% loss, 86.7% at 1% loss, and 73.3% at 2% loss. It leaves 80-83 failed sagas per run, with committed forward records that no compensation ever reverts. It also leaves about a hundred sagas per run that do not have a final event. Its throughput collapses from 1,079 to 146 sagas per second at 0.5% loss because clients must wait for the 3 s timeout. By contrast, Orchestration and ASO-Saga also experience a loss of throughput due to the 250 ms retry timeout, but ASO-Saga maintains higher throughput at all loss rates. The experiment also confirmed that the guarantees of consistency are not relaxed by parallel compensation, that orchestration makes the approach resilient, and that recovery time no longer increases with the saga's length.

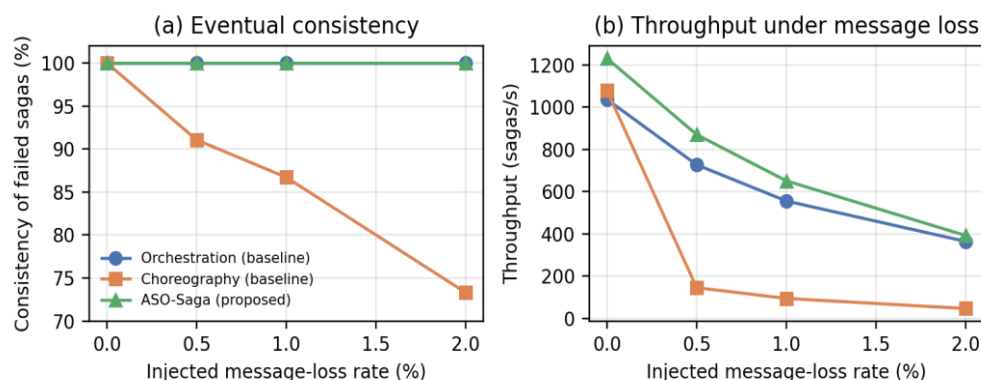


Figure 6. (a) Consistency of failed sagas and (b) throughput under injected message loss (25 virtual users, 40% error rate).

4.6 Ablation study

The benefits of each mechanism at 25 virtual users and a 40% error rate are shown in Figure 7 and Table 3. Disabling parallel levels, for example, increases median latency from 20.4 ms to 23.6 ms (an increase of 16%) and decreases throughput by 7%. This is confirmed by the fact that using batched compensation (that is, the sibling-compensation cost in Equation (3) disappears, which we measure directly) gives a recovery time of 4.7 ms; without batched compensation, recovery time rises from 7.3 ms to 10.4 ms (a 43% slowdown), but latencies are barely affected. Disabling hybrid dispatch increases the median latency by 8% because the notification step goes back to being a command with a reply and a log write. Each mechanism is targeting a different metric. This full framework is optimal for latency with the minor overhead of recovering after a failure occurred in a parallel (lower) level.

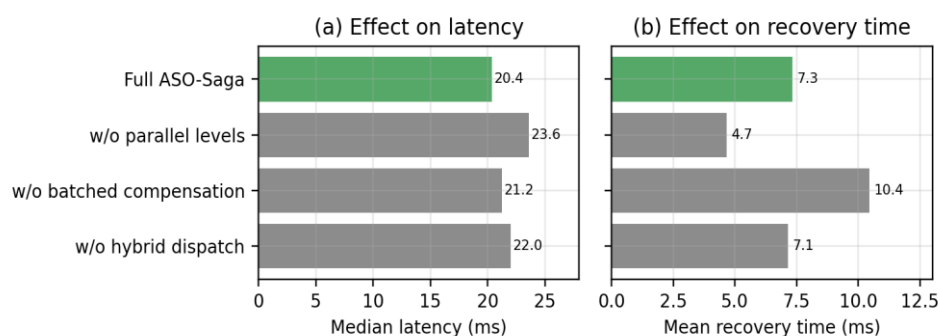


Figure 7. Ablation of ASO-Saga mechanisms with 25 virtual users at a 40% error rate: (a) median latency and (b) mean recovery time.

Table 3. Summary of measured results (throughput and latency at 10 virtual users; recovery at 8 services; consistency at 2% message loss).

Metric	Orchestration	Choreography	ASO-Saga	Change vs. best baseline
Throughput (sagas/s)	432	438	552	+26%
Median latency (ms)	22.9	22.6	17.1	-24%
p95 latency (ms)	30.0	30.0	25.4	-15%
Mean response time, 10% error, 25 VUs (ms)	28.3	28.3	23.8	-16%
Recovery time, 8 services (ms)	23.5	23.4	4.0	-83%
Recovery time, 5-step saga, 40% error (ms)	5.5	5.9	7.4	+35% (sibling compensation)
Consistency of failed sagas, 2% loss (%)	100.0	73.3	100.0	equal to orchestration
Messages per successful saga	10	5	9	-10% vs. orchestration

4.7 Discussion

This definitely indicates that the classical Saga pattern did not use all the performance offered. The increase in performance is because each step can be executed concurrently. This is consistent with Gan et al. (2019)'s finding that the latency of microservice systems is determined by the longest dependency chain: the recovery gain is caused by the fact that compensation is a set of idempotent operations that can be sent in bulk, corresponding with the event-log mechanisms described in Overeem et al. (2021). Compared to the isolation add-on of Daraghmi et al. (2022), the present protocol does not require an additional stateful component (e.g. a quota cache) and can be composed. Compared to strongly consistent protocols such as GRIT and 2PC* (Fan et al. 2019; Zhang et al. 2019), ASO-Saga does not lock services, and enjoys service autonomy. It sacrifices serialisable consistency for eventual consistency, which is already the trade-off enterprises make when they adopt sagas (Laigner et al., 2021).

There are three limitations to note, however: the prototype does not accurately model business logic due to fixed waits and runs on one CPU core. Thus, the absolute numbers and saturation point are environment specific but the comparison between the strategies is not, since they utilize the same infrastructure. Further, the parallel execution strategy benefits from a wider saga graph, since failure inside a parallel level incurs extra compensation. Second, we considered sagas whose most fragile step has no independent sibling, which could only benefit from batched compensation and hybrid dispatch. Third, coordinator failures and network partitions were not injected beyond message loss. In implementation, the framework may be adopted in incremental fashion since only idempotent compensating actions need be exposed by services. The reduced critical path and length-independent recovery also lead to lower response times for most latency-sensitive enterprise workflows.

5. CONCLUSION

This paper proposed ASO-Saga, an adaptive orchestration framework that runs independent Saga steps on different layers in parallel, chooses a coordination style for each step and batches compensations, running a multi-process prototype with one database per service. Compared to sequential orchestration and event choreography for the same workload and error rates as the prototype, ASO-Saga improved throughput 15-32% and decreased median latency 18-27% in the range of 1 to 25 virtual users. It has also been shown to outperform earlier protocols, maintaining the lowest response time from 10% to 70% error rates and reducing backward-recovery time by 83% for eight-service sagas. It maintains full consistency in the event of message loss and choreography lost at worst 27% of failing sagas. The ablation study shows that parallelism drives the latency gain and batched compensation drives the recovery gain, and the failures inside a parallelism level incur extra compensation. Future work will port the prototype to multi-core containerised dispatch backed by Kafka and PostgreSQL, add coordinator failover and partition injection, and combine parallel execution with isolation techniques of the recent work.

REFERENCES

- [1] Blinowski, G., Ojdowska, A., & Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10, 20357–20374. <https://doi.org/10.1109/ACCESS.2022.3152803>
- [2] Daraghmi, E., Zhang, C.-P., & Yuan, S.-M. (2022). Enhancing Saga pattern for distributed transactions within a microservices architecture. *Applied Sciences*, 12(12), 6242. <https://doi.org/10.3390/app12126242>
- [3] Di Francesco, P., Lago, P., & Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150, 77–97. <https://doi.org/10.1016/j.jss.2019.01.001>
- [4] Fan, P., Liu, J., Yin, W., Wang, H., Chen, X., & Sun, H. (2020). 2PC*: A distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform. *Journal of Cloud Computing*, 9(1), Article 40. <https://doi.org/10.1186/s13677-020-00183-w>
- [5] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., ... Delimitrou, C. (2019). An open-source benchmark suite for microservices and their hardware–software implications for cloud and edge systems. In *Proceedings of the 24th International Conference on Architectural Support for*

- Programming Languages and Operating Systems (ASPLOS '19) (pp. 3–18). ACM. <https://doi.org/10.1145/3297858.3304013>
- [6] Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., & Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3), 24–35. <https://doi.org/10.1109/MS.2018.2141039>
- [7] Kraft, P., Li, Q., Zhou, X., Bailis, P., Stonebraker, M., Yu, X., & Zaharia, M. (2023). Epoxy: ACID transactions across diverse data stores. *Proceedings of the VLDB Endowment*, 16(11), 2742–2754. <https://doi.org/10.14778/3611479.3611484>
- [8] Laigner, R., Zhou, Y., Salles, M. A. V., Liu, Y., & Kalinowski, M. (2021). Data management in microservices: State of the practice, challenges, and research directions. *Proceedings of the VLDB Endowment*, 14(13), 3348–3361. <https://doi.org/10.14778/3484224.3484232>
- [9] Limón, X., Guerra-Hernández, A., Sánchez-García, A. J., & Pérez Arriaga, J. C. (2018). SagaMAS: A software framework for distributed transactions in the microservice architecture. In *Proceedings of the 6th International Conference in Software Engineering Research and Innovation (CONISOFT)* (pp. 50–58). IEEE. <https://doi.org/10.1109/CONISOFT.2018.8645853>
- [10] Overeem, M., Spoor, M., Jansen, S., & Brinkkemper, S. (2021). An empirical characterization of event sourced systems and their schema evolution—Lessons from industry. *Journal of Systems and Software*, 178, 110970. <https://doi.org/10.1016/j.jss.2021.110970>
- [11] Rudrabhatla, C. K. (2018). Comparison of event choreography and orchestration techniques in microservice architecture. *International Journal of Advanced Computer Science and Applications*, 9(8), 18–22. <https://doi.org/10.14569/IJACSA.2018.090804>
- [12] Soldani, J., Tamburri, D. A., & Van Den Heuvel, W.-J. (2018). The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146, 215–232. <https://doi.org/10.1016/j.jss.2018.09.082>
- [13] Štefanko, M., Chaloupka, O., & Rossi, B. (2019). The Saga pattern in a reactive microservices environment. In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)* (pp. 483–490). SciTePress. <https://doi.org/10.5220/0007918704830490>
- [14] Waseem, M., Liang, P., & Shahin, M. (2020). A systematic mapping study on microservices architecture in DevOps. *Journal of Systems and Software*, 170, 110798. <https://doi.org/10.1016/j.jss.2020.110798>
- [15] Zhang, G., Ren, K., Ahn, J.-S., & Ben-Romdhane, S. (2019). GRIT: Consistent distributed transactions across polyglot microservices with multiple databases. In *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE)* (pp. 2024–2027). IEEE. <https://doi.org/10.1109/ICDE.2019.00230>
- [16] Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., & Ding, D. (2021). Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2), 243–260. <https://doi.org/10.1109/TSE.2018.2887384>