A Benchmark for Data Management Challenges in Microservices [Experiment, Analysis & Benchmark]

Rodrigo Laigner
University of Copenhagen
Copenhagen, Denmark
rnl@di.ku.dk

Zhexiang Zhang
University of Copenhagen
Copenhagen, Denmark
gjm699@alumni.ku.dk

Yijian Liu
University of Copenhagen
Copenhagen, Denmark
liu@di.ku.dk

Leonardo Freitas Gomes
Amadeus
Nice, France
leonardo.gomes@amadeus.com

Yongluan Zhou
University of Copenhagen
Copenhagen, Denmark
zhou@di.ku.dk

ABSTRACT

Microservice architectures emerged as a popular architecture for designing scalable distributed applications. Although microservices have been extensively employed in industry settings for over a decade, there is little understanding of the data management challenges that arise in these applications. As a result, it is difficult to advance data system technologies for supporting microservice applications. To fill this gap, we present *Online Marketplace*, a microservice benchmark that incorporates core data management challenges that existing benchmarks have not sufficiently addressed. These challenges include transaction processing, query processing, event processing, constraint enforcement, and data replication. We have defined criteria for various data management issues to enable proper comparison across data systems and platforms.

After specifying the benchmark, we present the challenges we faced in creating workloads that accurately reflect the dynamic state of the microservices. We also discuss implementation issues that we encountered when developing *Online Marketplace* in state-of-the-art data platforms, which prevented us from meeting the specified data management requirements and criteria. Our evaluation demonstrates that the benchmark is a valuable tool for testing important properties sought by microservice practitioners. As a result, our proposed benchmark will facilitate the design of future data systems to meet the expectations of microservice practitioners.

PVLDB Reference Format:

Rodrigo Laigner, Zhexiang Zhang, Yijian Liu, Leonardo Freitas Gomes, and Yongluan Zhou. A Benchmark for Data Management Challenges in Microservices [Experiment, Analysis & Benchmark]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Microservice architecture has emerged in the last decade as a popular architectural style in industry settings. This style promotes the decomposition of an application into independent microservices with associated private states. From an organizational point of view, these principles allow different teams to manage and evolve their own modules independently. At the same time, it enables new modules to be introduced and deprecated modules to be removed without impacting the application as a whole. From a technological point of view, each module can be independently deployed on distributed computational resources, allowing for failure isolation and high availability. Meanwhile, the message-based communication paradigm serves as a powerful abstraction for triggering tasks in remote microservices, facilitating data replication among microservices, and enabling failure recovery by replaying past events [14].

Despite the benefits of the decoupled design, a recent study [16] demonstrates that practitioners encounter several challenges when trying to meet data management requirements in this architecture, including:

(i) **Ensuring all-or-nothing atomicity.** The asynchronous and non-blocking nature of messages and the lack of interoperability across different data stores make distributed commit protocols difficult to implement, leading developers to either encode their own or eschew the use of synchronization mechanisms.

(ii) **Implementing efficient and consistent data processing.** While data is scattered across microservices, some workloads require querying and joining data belonging to different microservices. Therefore, developers often need to implement querying functionalities at the application layer that should belong in the database layer.

(iii) **Ensuring data replication correctness.** In order to reduce the expenses associated with querying data from remote microservices, developers often resort to caching or replicating data by subscribing to events generated by other microservices. However, as these events can arrive in any order, developers face challenges in maintaining consistent replication.

(iv) **Enforcing cross-microservice data integrity constraints.** As the application is functionally partitioned, data integrity constraints can span multiple microservices. This creates major challenges in constraint enforcement.

(v) **Ensuring correct event processing order.** Developers leverage application-generated events to implement event-based microservice workflows. However, due to the asynchronous nature of events, guaranteeing the correct processing order can be challenging. This is because events can arrive out of order, late, or even duplicated. Such scenarios pose a major issue when the application logic is sensitive to the processing order of events.

In short, microservices are designed to function independently, but in practice, they often rely on each other's data and functionality to complete a workflow. As a result, it is essential to benchmark microservices in a way that accurately reflects the needs of practitioners. Unfortunately, existing microservice benchmarks do not fully capture these real-world requirements [10, 16, 29]. For example, DeathStarBench [10] does not consider event-driven architecture, nor does it specify what data invariants and transactional guarantees are necessary.

To bridge this gap, we propose *Online Marketplace*, a novel microservice benchmark containing eight microservice types, ten event types, four types of business transactions, and three query types that reflect the key data management tasks pursued by practitioners, such as transaction processing, data replication, consistent queries, event processing, and data constraint enforcement. To reflect the data management challenges mentioned above, we prescribe seven data management criteria that a data management platform should meet, including functional decomposition, resource isolation, data consistency, and data integrity of microservices. These criteria are meant to embrace the complex nature of deployments found in industry settings and facilitate conducting a fair comparison between different systems on the same basis. To our knowledge, *Online Marketplace* is the first microservice benchmark that embraces core data management requirements sought by microservice practitioners.

Based on the definition of *Online Marketplace*, we further developed a data generator and a benchmark driver. The latter can continuously generate and submit transactions to an implementation of *Online Marketplace*. A challenge of implementing the driver is generating transactions at runtime that coherently reflect the dynamic application state, for example, the latest product prices and product versions. Querying the runtime application state imposes an unnecessary and prohibitively expensive workload on the system. To address this problem, we developed a stateful driver, which manages a consistent mirror of some application data and generates coherent transaction inputs.

Online Marketplace can benchmark various systems, platforms, and frameworks for developing data-intensive microservices. We verify the applicability of our benchmark by implementing *Online Marketplace* on two different platforms, Orleans and Statefun, which are designed for event-driven, distributed stateful applications. We conducted experiments to measure how these competing platforms perform under different workload scenarios. During the implementation of *Online Marketplace* on these state-of-the-art platforms, we encountered several limitations that prevented us from fulfilling some of the data management functionalities and criteria defined by *Online Marketplace*. Meanwhile, our results show that our benchmark can effectively stress the performance of the platforms and reveal performance and functionality issues. As a result, our proposed benchmark will facilitate the design of futuristic data systems that can fully meet the expectations of microservice practitioners, who are an important part of the database user community.

2 THE ONLINE MARKETPLACE BENCHMARK

In this section, we present *Online Marketplace*, a benchmark based on a marketplace platform that supports sellers offering a variety of products to customers, managing aspects related to stock, payment, and shipment of goods. It is designed to reflect the architectural design of microservice applications while emphasizing the challenges of data management in this architecture.

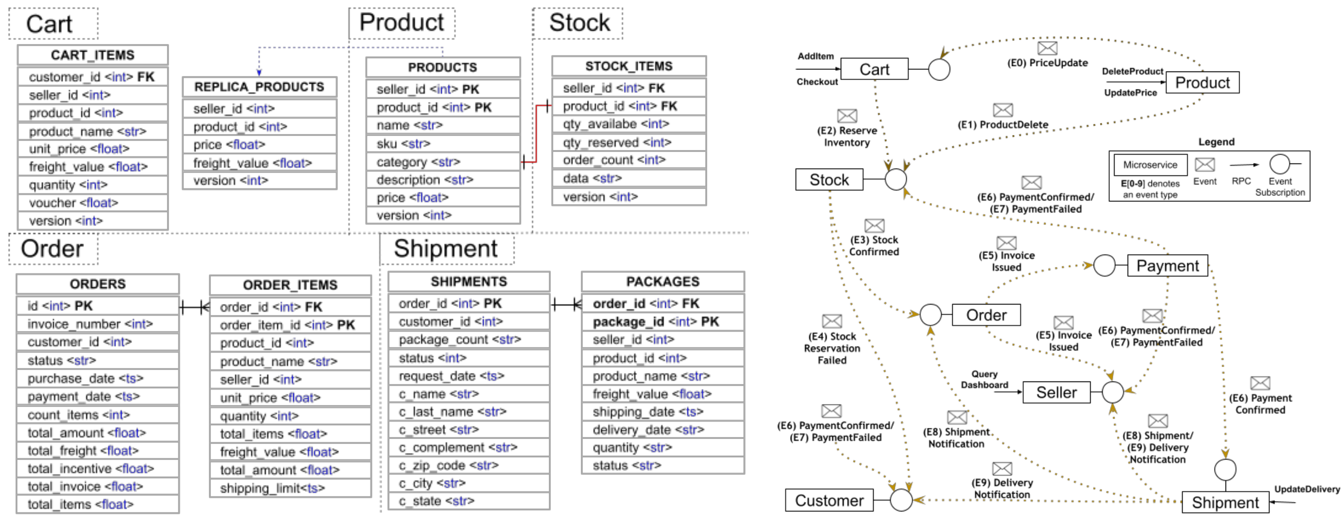
2.1 Application Scenario

Cart Management. Customers shop in *Online Marketplace*by navigating and selecting products from a catalog. A customer session is linked to a *cart*, with operations involving adding, removing, and updating items (e.g., increasing the quantity). A customer can only have one active cart at a time. When requesting a checkout, the customer must include a payment option and a shipment address, which are assembled together with the cart items and submitted for processing. On the other hand, customers may also abandon their carts before submitting the checkout request.
Catalog Management. Each product is offered by a particular seller. Sellers are responsible for managing their products and associated stock information. They may replenish products and adjust their prices.
Customer Checkout. Checkout requests follow a chain of actions: stock confirmation, order placement, payment processing, and shipment of items. Upon submission of an order, if one or more items do not have sufficient stock, the checkout will still proceed with the available items.
Payment Processing. Upon stock confirmation, the payment details provided by the customer are used to make a payment request to an external payment service provider (PSP) [12]. A payment can fail for two reasons: rejection of the transaction by the provider or impossibility of contacting the payment provider.
Order Shipment. Upon approval of payment, the shipment process starts. For each seller present in an order, a shipment request is created. A shipment request includes the items present in the order. Each item is reflected as a package that should be delivered to the customer at some point.
Package Delivery. Whenever a package is delivered, both the corresponding customer and seller are notified. When all packages of an order are delivered, the order is considered completed.

2.2 Microservices

Traditionally, a microservice-based application is often made up of independent services, each deployed in a container on an OS-level virtualized platform such as Docker [16]. However, the emergence of programming frameworks for designing distributed applications, such as Orleans [3] and Statefun [24], provides programming models that allow microservices to be implemented using higher-level abstractions, such as actors and functions, respectively.

To allow this benchmark to support the gamut of microservice implementations, we withdraw ourselves from defining an architectural blueprint, but rather, we focus on describing the expected



(a) Partial Data model **(b) Events**

Figure 1: Marketplace Microservice Application

independent components that must compose the benchmark application.

In this sense, in the following paragraphs, we take advantage of Figure 1 to clarify the interactions, APIs, and the data model prescribed for each microservice. It is worth noting that we use the term "events" as a communication abstraction for microservices, but these can also be framed as any message payload asynchronously delivered to a microservice. In addition, the event identifiers in Figure 1(b) do not imply a particular order in which the microservices exchange events.

Furthermore, while we specify the microservices' state and the queries using the relational model, they can also be specified through other data and query models as long as the same functionalities can be achieved.

Cart. The *Cart* microservice allows the customer to manage the products that ought to compose a checkout order. It does so through the following APIs:

- AddItem. Add a product with an associated quantity to a customer's cart.
- RemoveItem. Remove a product from a customer's cart.
- Checkout. Submit a customer's cart for checkout and seal the cart.

To manage a customer's cart, two relations are used: `carts`, to track the status of a customer session, and *cart_items*, to store the items that customers are willing to buy. Moreover, the *replica_products* relation represents a partial replica of the *product* table belonging to the `Product` microservice. The table is only updated through `ProductUpdate` events received from the `Product` microservice (represented by a dotted blue arrow in Figure 1(a)). The replication semantics are discussed in Section 3.3.

When a checkout request is received, `Cart` ensures the correctness of the customer's cart by matching the cart items to the replicated product information and looking for products that had their price updated. In this case, the checkout is denied and the new prices are informed to the user. Otherwise, the customer's cart items are assembled into a `ReserveInventory` event and published for asynchronous processing. Once the event is published, the cart is sealed, allowing the customer to initiate a new cart and execute a series of cart operations.

Product. The *Product* microservice manages the catalog of products. It performs operations over a single relation: `product`. For every update operation triggered via the following API, a corresponding event is generated for downstream processing: E0 for UpdatePrice and E1 for DeleteProduct.

- GetProduct. Retrieve a product based on a seller and a product identifier.
- DeleteProduct. Marks a product as unavailable to customers.
- UpdatePrice. Update the price of a product.

Stock. The *Stock* microservice manages the inventory through a single relation: *stock_item*. The following API is provided:

- GetStockItem. Retrieve the information of a stock item.
- AddStockItem. Add a stock item with an associated quantity available.
- ReplenishStock. Replenish stock for a given item.

To update the inventory, the *Stock* microservice processes four events: `ReserveInventory`, `PaymentConfirmed`, `PaymentFailed`, and `ProductDelete`. Every inbound event leads to updates in one or more *stock_item*' tuples (depending on the number of items present in the checkout).

On processing a `ProductDelete` event, the *Stock* microservice marks the respective *stock_item* as unavailable, meaning that future checkouts (triggered by `ReserveInventory` event) on this item cannot be carried out (represented by a red connector in Figure 1(a)).

Processing the `ReserveInventory` event leads to marking the stocks of some items (subject to availability) presented in the customer's cart as reserved. The result of processing `ReserveInventory` might lead to the generation of two events: (i) `StockConfirmed` if at least one product has been reserved, and; (ii) `ReserveStockFailed` if no product requested by the customer is available. In both cases, the respective items are assembled together in the respective event for downstream processing. Lastly, PaymentConfirmed confirms the reservation and PaymentFailed would lead to withdrawing the corresponding stock reservations.

Although at first sight the differences between Product and Stock may seem blur, it is important to highlight they provide distinct functionalities in the application. While the Product is responsible for the correctness of the catalog data, including but not limited to the characteristics of a product like name, category, seller, and price, the Stock microservice ensures the integrity of the inventory. It is important to address such design reflects real-world deployments [25].

Order. The *Order* microservice manages customer orders' data. It does so by maintaining four relations: *order*, *customer_orders*, *order_items*, and *order_history*.

The *order* relation contains general data about an order, including but not limited to customer, amounts, etc. *customer_order* relation tracks the number of orders requested by each customer, which is used to form the invoice number. Order items relate to the products requested in checkout and *order_history* tracks updates to an order (invoiced, paid, shipped, completed).

Order's relations are updated upon processing the following events: StockConfirmed, ShipmentNotification, PaymentConfirmed, PaymentFailed.

Upon receiving a StockConfirmed event, the amount to charge the customer is calculated (including freight and discounts) and an invoice number is generated, resulting in a InvoiceIssued event. During this processing, all four relations have tuples created (in case it is the first customer order, otherwise *customer_order* tuple is updated).

Afterwards, there are two moments where a customer's order status is updated: after a payment and after shipment updates. Both trigger writes to order and *order_history* relations. In this case, the order of event processing here plays a role on maintaining the notion of time progress for individual orders.

Payment. The *Payment* microservice is responsible for managing payment data. Two relations are used to track an orders' payment data: *payment* and *card_payment*.

Payment processing starts through processing an InvoiceIssued event. Every credit applied to the order, namely, discounts and the payment option chosen by the customer at the time of the checkout (credit/debit card or a bank slip) are stored in *payment* relations and in case of a card payment, a *card_payment* tuple is also inserted.

As part of the payment processing, it may be necessary to coordinate with an external payment provider to ensure the provided payment method is valid. In this sense, *Payment* microservice relies on an idempotent API offered by the external payment provider. In case the payment microservice fails right after confirming a payment, subsequent attempts (after recovering from failure) to charge the customer does not incur in duplicate payments.

Processing an invoice can lead to two outbound events: PaymentConfirmed and PaymentFailed. The PaymentConfirmed event contains all items present in the paid invoice whereas the PaymentFailed is a simple payload containing the invoice number so the *Order* microservice can update its state accordingly.

Shipment. The *Shipment* microservice manages the lifecycle of a shipment processing, including shipment provision and delivery of goods. Two relations are maintained: *shipment* and *package*.

A shipment process starts by processing a PaymentConfirmed event, creating a delivery request for each order item and confirming all goods have been marked for shipment by producing a ShipmentNotification with status 'approved'.

At a later moment, through the *UpdateShipment* API, a set of shipments and associated packages tuples are updated. Each package delivered leads to the generation of a respective DeliveryNotification event and, in case all items that form a shipment are delivered, a ShipmentNotification with status 'concluded' is also emitted.

Customer. The *Customer* microservice manages customer data, including their home address, contact information, and credit card. Besides, statistics about the customer are updated through processing the following events: PaymentConfirmed, PaymentFailed, ReserveStockFailed, and DeliveryNotification.

Seller. The *Seller* microservice manages seller-based marketplace data. It does so by processing events and transforming them into seller-centric data. Three relations are found in *Seller's* schema: *seller*, *order_entry*, and *order_entry_details*.

The relation *order_entry* represents an order item, but from the perspective of a seller. In this sense, amounts related to the item (i.e., discount, freight, total) are calculated. Basically every order (derived from the InvoiceIssued payload) is transformed into *N* *order_entry* tuples, where N is the number of items corresponding to a seller. *order_entry_details* relation contains data agnostic to sellers, such as payment method used and total order amount. Besides, the relation *order_entry* is updated through processing the following events: PaymentConfirmed, PaymentFailed, ShipmentNotification, and DeliveryNotification.

2.3 Workload

In this section, we describe the *Online Marketplace*workload, namely, the business transactions and continuous queries the application must cope with.

2.3.1 Business Transactions. To realize the application scenario, we describe four business transactions that reflect different complexities in terms of number of involved microservices and number of events processed. In Section 3, we discuss how transactions possess different properties.

Customer Checkout. It starts in *Cart* microservice through processing a Checkout request. It involves *Cart*, *Stock*, *Order*, *Payment*, *Shipment*, *Seller*, and *Customer* microservices. A success path involves the following events: E2 - E3 - E5 - E6 - E8. There are two error cases: (i) when payment fails, then from E5 we have E7 as the last; and when no single item from the cart is available in stock: E2 - E4. Although not shown in the Figure 1 due to space constraints, a success path also involves E6 being processed by *Order* microservice.

Price Update. To enable the partial replication of products in the *Cart*, upon processing a UpdatePrice request and updating its private state accordingly, the *Product* microservice generates E0 and sends it to the *Cart* microservice. By processing E0, the *Cart* microservice applies the new price to its corresponding product replica.

Product Delete. To simulate making a product unavailable to customers, we pick a seller and a corresponding product (both from a distribution) and we set the product as disabled. To maintain the total number of products, thus avoiding anomalies in the distribution, we replace the deleted product with another one.

This operation is carried out through processing a `DeleteProduct` request in *Product* microservice. Upon updating its private state accordingly, *Product* generates E1 and sends it to the *Stock* microservice.

Update Delivery. To simulate the delivery of goods, we pick the first 10 sellers with uncompleted orders (i.e., at least one package has not been delivered yet) on chronological order and we set their respective order's packages as delivered.

The idea is that oldest packages are progressively delivered as more update delivery transactions are submitted to the system. For maintenance of statistics about customer and orders, the following events are generated along with the transaction: `E9` is generated by *Shipment* for every package delivered and sent to the Seller and Customer microservices; and `E8` is generated by *Shipment* when all packages of a shipment are delivered and sent to the *Order* microservice.

2.3.2 Continuous Queries. Continuous queries over event streams constitute an emergent trend in microservice applications. As event payloads produced by different microservices often contain state information [9], it is possible to (indirectly) access data from multiple microservices without breaking their encapsulation. In other words, continuous queries can be built based on events without resorting to (synchronously) pulling data from each required microservice.

In this section, we describe three different types of continuous queries covering important concerns in *Online Marketplace*. To specify the queries, we use the syntax of Materialize [19], a streaming database. Due to space constraints, we list one continuous query along this section and the others can be found in our extended version [15].

Online Query #1: Seller Dashboard

The business scenario encountered in a marketplace requires sellers to have and end-to-end overview of the operation in real time to identify trends, such as the popularity of products, and to support decision making, such as when to increase product prices.

Query Description: In this query, we want to determine the total financial amount of ongoing orders by seller. Assuming there is a stream (Listing 1) that filters out concluded and failed orders, one could implement this continuous query as shown in Listing 2.

Listing 1: Ongoing order entries base query

```
CREATE MATERIALIZED VIEW order_entries
SELECT order_id, seller_id, product_id, ...
FROM InvoiceIssued as inv
LEFT JOIN ShipmentNotification as ship ON ship.order_id =
      inv.order_id
LEFT JOIN PaymentFailed as pay ON pay.order_id = inv.order_id
WHERE ship.status != 'concluded' AND pay.order_id IS NULL
GROUP BY seller_id
```

Listing 2: Ongoing orders aggregation per seller

```
SELECT seller_id, COUNT(DISTINCT order_id) as count_orders,
      COUNT(product_id) as count_items, SUM(total_amount) as
      total_amount, SUM(freight_value) as total_freight,
      SUM(total_items - total_amount) as total_incentive,
      SUM(total_invoice) as total_invoice, SUM(total_items) as
      total_items FROM order_entries
GROUP BY seller_id
```

In addition, the seller dashboard output must also discriminate the records that compose the aggregated values computed in the above query. In this sense, it should also present to the user the following query result:

Listing 3: Ongoing orders discriminated

```
SELECT * from order_entries
WHERE seller_id == <sellerId>
```

Online Query #2: Cart Abandonment. A popular use case arising from a series of customer interactions is cart abandonment [18]. A cart is considered abandoned in two cases: (i) prior to checkout submission, and (ii) upon a failed payment processing, if no customer checkout re-submission is identified.

Query Description: In this query, we want to find the cart checkouts that have either failed via stock reservation or payment attempt and have not been involved in a new checkout attempt within the next 10 minutes after the failure. Upon detecting an abandoned cart, a `CartAbandoned` event is generated for both *Cart* and *Customer* microservices. The query specification can be found in our extended version [15].

Online Query #3: Low Stock Warning

To assist sellers, marketplaces usually monitor products' stock proactively to notify sellers about low inventory and ultimately refrain customers from experiencing unavailability of products.

Query Description: In this query, we want to find the products (and their respective sellers) that are likely to face unavailability in case no replenishment is provided in the near future.

The search is based in the following criteria: The average number of items requested per week in the last month, independently of the result of the reservation and payment, is higher than the present stock level. The *Low Stock Warning* continuous query specification can be found in our extended version.

2.4 Wrapping it Up

The application represents a real-world Marketplace scenario and the clients (Customers, Sellers, and Delivery Company) that interact with. Similarly to the state of the practice, the application provides a balance in terms of how events are produced and processed, and how data is processed across different microservices. For example, while *Cart* and *Product* react to no events, acting as user-facing services by responding to synchronous requests, *Customer* and *Seller* microservices only react to events, producing none. Besides, *Stock* microservice requires isolation among different events being processed to ensure stock correctness while *Payment* can safely disregard ordering semantics on processing payments.

The table 1 presents an overview of the number of microservices accessed and the number of events exchanged per transaction. We refer to *AVG* as the average number of items from each seller per order. As we see in Section 4, this is defined by the number of items per order and the seller distribution.

Table 1: Transaction Overview

Business Transaction	# Microservices Accessed	# Events
Customer Checkout	7	10
Update Delivery	4	$\lfloor 10 * 2 * AVG \rfloor$
Update Price	2	1
Delete Product	2	1

3 DATA MANAGEMENT CRITERIA

In this section, we discuss the criteria that an implementation of *Online Marketplace* should meet. The criteria reflect key principles and challenges of data management in microservices. Some criteria have multiple levels, allowing the benchmark users to choose the most suitable one that fits their specific requirements. Furthermore, our explicit criteria specifications facilitate conducting fair comparisons between different systems on the same basis.

3.1 Functional Decomposition

Microservice applications require functional partitioning, i.e. application functionalities are decomposed into different microservices. There are two issues in achieving functional decomposition: (i) how microservices interact and (ii) how microservices manage states.

(i) Microservices interact with each other through asynchronous events to carry out functionalities across multiple microservices. For example, completing a cart requires a composition of functionalities across *Cart*, *Stock*, *Order*, *Payment*, and *Shipment* microservices. This composition of functionalities is enabled through the events they exchange.

(ii) State management is divided into two categories:

a. Direct data access: Each microservice can directly access its own data (encapsulation principle).

b. Indirect data access: Data within each microservice can only be accessed externally through predefined interfaces (e.g., seller dashboard) or be notified about state changes through events (e.g., *Cart* is notified about product updates). It should be noted that this criterion does not require data from different microservices to be stored in different databases.

3.2 All-or-nothing Atomicity

The business transactions must comply with all-or-nothing atomicity semantics. This criterion is crucial to guard against crashes or performance degradation leading to failures in the middle of a business transaction.

3.3 Caching or Replication

Section 2.1 describes the case of *Cart* subscribing to product updates with the goal of ensuring that checkouts do not contain outdated product prices. As the strategy of implementation varies system by system, either cache or replication can fulfill this requirement. In this sense, we prescribe four possible correctness semantics:

(i) Eventual. Updates (price update or product delete) are processed independently, disregarding the order that they are generated at the source, i.e. *Product*.

(ii) Causality at the object level. Updates on the same product are processed sequentially in accordance with the order in which they were performed at the source.

(iii) Causality across multiple objects. All updates made by the same seller must be applied to the cart in the same order they were applied at the source, achieving read-your-write consistency.

3.4 State Management Constraints

3.4.1 *Inter-microservices constraints.*

Refers to a constraint that cut across microservices.

Constraint #1: Cross-microservice referential integrity

Stock always references an existing product in *Product*. We take inspiration from the CAP theorem [11] to define two levels of consistency:

(i) Available System. A deleted product will eventually not be allowed to be reserved anymore. The transaction is considered committed after *Product* responds to the `Delete Product` request.

At this level, if there is a network error or failure of the microservices, the delete message (EY) may not reach *Stock*. That requires the inconsistency to be detected and resolved at a later point.

(ii) Consistent System. `Delete Product` request is committed only when both Product and Stock have committed, therefore Product and Stock are always consistent with each other. At this level, a deleted product will not be available for future checkout attempts.

Constraint #2: No duplicated checkouts. A customer cart belonging to a customer session must not be checked out more than once. This can be safeguarded by having mutual exclusion in each cart. Another way to ensure this constraint is by assigning a customer session ID to each new customer session. The ID is included in the payload of each checkout request submitted to *Order*. Upon receiving it, *Order* ensures the same cart checkout does not lead to duplicate order processing.

3.4.2 *Intra-microservice constraints.*

These include invariants that can be enforced by relying on only local states. They can be achieved through appropriate isolation levels.

Constraint #3: No overselling of items. In *Stock* microservice, both available and reserved quantities cannot fall below zero for every item. In addition, the reserved quantity must never be higher than the available quantity.

Constraint #4: Maintenance of customer statistics. The system tracks successful and failed payments and deliveries through increments of numbers. An appropriate isolation level is required to ensure that the increment in *Customer* is not missed.

Constraint #5: Linearized product updates. Updates on each individual product's price or version in the *Product* state must be linearized.

Constraint #6: Concurrent Update Delivery transactions must execute in isolation. Since several events can be generated during a *Update Delivery* transaction, it is essential that concurrent *Update Delivery* transactions do not operate on the records of the same shipments and packages. The purpose is to avoid emitting duplicated `ShipmentNotification` and `DeliveryNotification` events.

Constraint #7: Stock operations triggered by events must execute in isolation. Processing events in *Stock* requires either serializable isolation or exclusive locks on the items contained in the event payload.

3.5 Event Processing Constraints

Event Order. Event processing order impacts *Order* and *Seller* dashboard. A microservice data platform can provide two levels of event ordering guarantee.

Unordered: `PaymentConfirmed` and `ShipmentNotification` events are processed arbitrarily, possibly leading to violating the natural order of a payment occurring before a shipment.

Causally Ordered: The `PaymentConfirmed` event must always precede shipment events. Similarly, a `ShipmentNotification` with the status 'approved' must always precede the corresponding `ShipmentNotification` with the status 'concluded'.

Consistent Snapshot for Continuous Queries. For the two concurrent queries of the seller dashboard to be consistent with each other, their results should reflect the same snapshot of the application state.

Event Delivery. The data platform can provide three levels of event delivery guarantees, namely at-most-once, at-least-once, and exactly-once delivery. The delivery guarantee has impacts on how to implement continuous queries and business transactions in *Online Marketplace*to achieve correctness. At-least-once and at-most-once delivery can make queries inaccurate. For example, while at-most-once delivery provides a lower bound on the profits in the *Seller Dashboard* without replaying lost messages, at-least-once can provide skewed results without accounting for duplicate events.

For transactions, at-least-once delivery requires that microservices must be idempotent to account for the possible duplicate event processing. On the other hand, there must be a timeout mechanism to abort transactions if at-most-once or exactly-once delivery is used to avoid any stalled transactions from blocking the other transactions.

3.6 Performance and Failure Isolation

The functional decomposition of the application allows microservices to operate independently, therefore, providing benefits from the isolation of resources assigned to each microservice. Microservice implementations of *Online Marketplace*can achieve isolation in two tiers:

(i) Each microservice's code/logic is executed on different computational resources, achieving performance and fault isolation at the application tier. This minimizes the impact of resource usage and failures between different microservices.

(ii) In the database tier, database operations do not interfere with each other in terms of performance and failure, achieved through isolated resource allocation to databases.

3.7 Logging

Audit logging is a critical concern in applications today as it allows developers to track application events like user activities (e.g., checking out a cart) and state updates [7]. The log recorded during application execution serves as an audit trail, aiding developers in troubleshooting faults and verifying compliance with prescribed business rules.

Logging records of operations is even more pressing in distributed systems, such as microservices. The substantial exchange of events and the complex interplay of independent components make it challenging to reason about errors and failures involving multiple asynchronous microservices.

In *Online Marketplace*, there are two key events related to logging historical records of operations: `ShipmentNotification` and `PaymentFailed`.

(a) Upon `ShipmentNotification` with `status` 'completed' or `PaymentFailed`, *Order* logs all records associated with such an order, in particular the relations *order*, $order\_items$, and $order\_history$.

(b) Upon `ShipmentNotification` with `status` 'completed', *Seller* logs all records associated with such shipment, in particular the relations $order\_entry$ and $order\_entry\_details$.

(c) As part of the emission of a `ShipmentNotification` with `status` 'completed', *Shipment* logs all records associated with such shipment, in particular those in the relations *shipment* and *package*.

(d) As part of payment processing, *Payment* logs payment records independently of the outcome (success or failure), particularly the relations *payments* and $payment\_cards$.

4 DATA AND WORKLOAD COMPOSITION

The workload submitted to *Online Marketplace* can be adapted to fit particular needs. In the following, we present the different configuration parameters that can de defined for experiments.

Data Population. The state of some of the microservices require initialization prior to workload submission. The following procedure is expected, where X, Y, and Z are configuration parameters:
(i) X number of customers are inserted into *Customer* microservice;
(ii) Y number of products are inserted into *Product* microservice;
(iii) Y number of stock items are inserted into *Stock* microservice. Each $stock\_item$ tuple must refer to an existing product (through *seller_id* and *product_id* columns) in *Product* microservice;
(iv) Z number of products per seller leads to (X/Z) seller tuples inserted into *Seller* microservice. In case of positive remainder, the last seller must own W products, where $W < Z$.

On the one hand, the number of customers should be large enough to accommodate the maximum amount of concurrent transactions running the system at a given time. On the other hand, the number is limited by the amount of memory available. Furthermore, the numbers of products and sellers should be large enough to not creating a lot of conflicts when running uniform distribution.

Optionally, benchmark users can populate records in other microservices to simulate preexisting data and introduce a degree of overhead in state accesses during the workload execution. We use the size factor to populate the database. A size factor S leads the *order* table to be initialized with S*1K tuples. In consequence, *payment*, and *shipment* follow the same size given their tuples refer to an existing order. For tables containing a foreign key attribute referencing the primary key of the above tables, we use the *scale factor* defined below to define the number of *order item* tuples per order and consequently, the number of *package* tuples per shipment.

Customer Checkout Transaction. The customer checkout offers different parameters in order to fulfill different application scenarios.

(i.) Number of items added to cart. For each cart, carts will have from 1 up to N items added, where N is defined via parameter. The number is chosen from a random distribution. N is the **scale factor** that determines the size of a transaction and its execution cost.

(ii.) The quantity per added cart item. A random value from 1 to a constant (e.g. 5), defined via parameter, is selected.
(iii.) Probability of cart abandonment. A parameter is defined prior to workload submission ranging from 0 to 100. A random number is selected and, if it falls below or is equal to the parameter defined, then the cart is abandoned before checkout submission.
(iv.) Probability of having a discount applied to a cart item. First, a uniform distribution is used to define whether an order will contain discounts. If the output number falls below or equal to the probability of having a discount, then a discount is selected next.

To define the discount to be applied, a random value from 1 to N is selected. N is the percentage of discount a customer gets for a product based on the observed price. Each discount leads to an additional record being recorded in *Payment* microservice state (relation *order_payment*) in case of a successful payment.
(v.) The payment method used. It governs how payment methods are selected. A uniform distribution is used to select either bankslip, credit card, or debit card. A credit card payment method leads to an additional record being written to payment's state (referring to relation *order_payment_card*).
(vi.) Probability of payment rejection. It governs the likelihood of a payment being rejected. A parameter is defined prior to workload submission ranging from 0 to 100. A random number is selected and, if it falls below or is equal to the parameter defined, then the payment gets rejected.
Ratio of Transactions. It defines the probability of submitting each business transaction and continuous query requests to the system. The transaction ratio can be tuned to specific goals and business scenarios for the benchmark users, such as query- or transaction-heavy scenarios.
Distributions. Distribution in the workload is centered on sellers and their products. For every operation involving a product (add cart item, product price update, and product delete), one has to pick first a seller and then proceed to pick a corresponding item from the seller's product keyset. In this way, Uniform and Zipfian distributions can be used interchangeably in two cases:
- Seller selection: for every product selection, one has to first pick a seller based on a defined distribution.
- Product selection: After a seller is previously selected, one picks a seller's product based on a defined distribution.

5 BENCHMARK METRICS & DRIVER

In this section, we discuss the benchmark driver, the program responsible for managing the life-cycle of experiments and collecting metrics as part of *Online Marketplace* application execution.

5.1 Metrics

We collect two metrics in this benchmark:
1. Throughput: The number of transactions processed per second, which include both business transactions that have successfully completed and continuous queries that have successfully returned a result update.
2. End-to-end latency: It is measured from the time a request (or an update event) is sent to when the sender (or query client) receives the transaction response (or the query result update).

5.2 Driver Functionalities

To manage the life-cycle of an experiment composed by: (i) data generation, (ii) data ingestion, (iii) data check, (iv) workload submission, (v) collection of results, (vi) report generation, and (vii) cleaning of states, we develop a benchmark driver in .NET.

A user specifies through a configuration file the workload requirements, including the concurrency level, number of products, number of sellers, experiment duration, ratio of transactions, cleaning procedures, as well as the target platform APIs in order for the driver to establish connections to a target platform and submit requests.

The driver parses configuration files dynamically and initializes the experiment lifecycle accordingly. The driver also allows the user to run a specific step, such as data generation, through an interactive menu. We envision a typical usage of the driver as follows:

A user usually starts with generating data. Sellers, customers, products, and stock items are generated synthetically and can be either stored durably in a database or kept in main memory. If data has been generated beforehand, the user can load the data from a specified database and move on to data ingestion, populating the microservices with initial data.

Upon completing data ingestion, the driver waits for a customized period before submitting transaction requests. The driver controls the maximum number of concurrent transactions running in the target platform by:
(i) Initializing a number of threads defined by the concurrency level. Each thread simulates a user interacting with the application;
(ii) Whenever a transaction result returns, the driver pulls a previously used thread from a thread pool and spawns a new transaction submission. Upon experiment completion, the driver computes the metrics of completed transactions and stores the result in a file.

It is noteworthy the driver expects a target platform to expose specific HTTP APIs (described in our extended version [15]), as it is commonly found in web services, in order to submit operations (e.g., add cart to item or add product) and transaction requests (e.g., checkout).

5.3 Driver Implementation Challenges

The *Online Marketplace* benchmark presents particular characteristics that necessitate caution on submitting transactions.
Simulating Customer Sessions. We start with the *Cart* microservice, which can be exemplified as a stateful operator. In the context of a customer session, *Cart's* state evolves by having sequential operations (i.e., add cart item) up to a point where the state is sealed (through a checkout request), then returning to the initial state. To this end, the workload submission must account for each customer session individually. In case there is an active customer session for customer `X`, there should be no concurrent driver's thread simulating the same customer. The reason is that concurrent operations on `X`'s cart state can interleave, making it difficult to maintain the workload distribution.

The driver enforces the above criteria by maintaining "idle" customers (those that have either not started any session yet or have completed a session) in a concurrent queue. A customer is pulled from the queue whenever a new customer session starts. Whenever a session terminates, the customer is enqueued again.

Managing Coherent Product Versions. As explained in Section 2.3.1, products could be deleted and replaced by new ones. In other words, new product versions are being generated online continuously. It is crucial to generate transactions referring to coherent product versions. For example, to generate a price update transaction, we need to make sure to refer to the current product version in *Product*'s state. As another example, on adding an item to a cart, we need to make sure cart items refer either to the current product version in *Product* or the version that preceded the current one at the time of this operation. The goal is to simulate the latest product version "seen" by a customer.

To obtain a coherent product version while constructing transactions, we have to know the application state. However, the benchmark driver should not query microservice states during transaction submission for two reasons: (i) Workload generation should be independent of the actual data platforms being used; (ii) Querying the state of microservices would introduce additional load that is not prescribed in the benchmark.

In order to generate correct transactions while not querying the microservices' states, the driver manages internally a consistent mirror of the *Product* microservice state to guarantee access to coherent product versions. The driver linearizes the submission of concurrent update requests to the same product; the compare-and-swap mechanism is used to decrease synchronization costs.

Whereas the picking of product versions for building a cart item runs concurrently with product updates so that customer and seller threads do not block each other.

Matching Transaction Requests to Asynchronous Results. It is common that platforms for building microservice applications provide results asynchronously (Section 5.2). In such an asynchronous system, the driver must track each submitted transaction and match it with the corresponding asynchronous transaction result to compute the metrics accurately.

To this end, each transaction request is assigned a timestamp and a unique ID. This ID is later used to match the request to a corresponding transaction result that is eventually received. To avoid threads blocking each other, ID generation is decoupled from transaction submission.

In conclusion, a stateful driver is necessary to provide correct transaction input.

6 EXAMPLE IMPLEMENTATIONS

To show how *Online Marketplace* can be used, we implement *Online Marketplace* in two competing platforms, Orleans and Statefun, both of which are designed to develop event-driven services with state management functionalities.

6.1 Orleans

Orleans is a framework that enables developers to build distributed stateful applications using the abstraction of virtual actors. This extended actor model, as described in [3], automatically allocates virtual actors in available resources, and releases them when they are no longer needed. This provides developers with location and life-cycle transparency, making it easier to manage resources. Additionally, actors are migrated from faulty computational resources to healthy ones in case of failures, ensuring that the application remains functional.

Table 2: Components Design

Actor/Function	Description
Cart	Models a customer's cart state and behavior
Customer	Models a customer's state and behavior
Seller	Models a seller's state and behavior
Product	Models a product's state and behavior
Stock	Models a stock item's state and behavior
Order	Models a unit of order processing for a single customer and associated state
Payment	Models a unit of payment processing for a single customer
Shipment	Models a unit of shipment processing for a disjoint group of customers and related state

Due to the single-threaded actor abstraction, developers are encouraged to decompose application functionalities into distinct actors to avoid a few specific virtual actors becoming the bottleneck. Inspired by the guidelines of Wang et al. [27], our design aim to maximize parallelism and minimize transaction latency by assigning *Online Marketplace* functionalities to different actors, as shown in Table 2. We implement *Online Marketplace* on Orleans 7.2.1, using both the default non-transactional Orleans API and the transactional API, referred to Orleans Transactions (TX) in the rest of the paper. To allow Orleans actors to be reachable from the driver, we deploy an HTTP server on top of the Orleans silo. The server is responsible for parsing and forwarding incoming requests to the appropriate actors, reporting back the transaction results via an HTTP interface.

6.2 Statefun

Statefun is a platform built on top of Flink for running distributed applications based on the concept of stateful functions. Each function manages its own logical state and reacts to incoming messages asynchronously. Each incoming message is processed sequentially, in a way similar to Orleans actors. As a result, applications can be designed by composing functions through messages. Statefun transparently allocates function calls across worker nodes that can be distributed across computational resources, freeing developers from handling faulty nodes and message retries.

We implement *Online Marketplace* on Statefun 3.3, and, given the resemblance of both programming models, we opted to model stateful functions with the same design as Orleans. To match the architectural design found in web services, we deploy Statefun with an HTTP ingress. Once the driver submits the transaction input, the ingress acknowledges the reception of the request and dispatches it to the appropriate function. We also deploy an HTTP egress to allow for the collection of transaction results that are eventually completed. The function that terminates a transaction sends a completion message to the egress. The egress operator stores the result and makes it available to clients. This design follows the Statefun documentation [23]. To execute Stateful Functions runtime, we follow the recommended deployment mode using docker images and the most performant execution style (embedded functions) [22].

6.3 Implementation Challenges

Upon implementing the *Online Marketplace*features in each platform, we encountered some limitations. We explain in the following how we mitigated them and the criteria that the target platforms can meet. Note that we only consider features that are natively supported by the framework, and use as few external systems as possible.

Replicating products. Given that both platforms do not provide indexing for the actor or function states, we cannot efficiently query which carts contain a particular product. Therefore, we opted not to implement this replication feature in the experiments.

Continuous queries. We do not use external stream processing systems in order to benchmark only the target platforms rather than the integration of multiple systems. Therefore, we opted to only implement the query dashboard, but not the continuous queries of cart abandonment and low stock warning. The reason is that the `order_entries` view can be maintained through conventional in-memory data structures and updated through application events, while the others require more complicated stream processing operators such as windowed aggregates and joins, which are not provided natively by the two platforms. This also indicates an opportunity for these platforms to improve their support of microservice applications.

Messaging delivery guarantees. Orleans provides an at-most-once delivery guarantee by default. Although Orleans can be configured to send retries upon timeout, we opted to capture the timeout exception and report to the driver that the transaction has been completed with an error. The reason is that, by enabling retries, the message may arrive multiple times, potentially introducing errors if the application is not idempotent.

On the other hand, Statefun manages state storage and message delivery in an integrated manner, such that in the presence of a delivery error, Statefun transparently retries the delivery up to a timeout and, upon that, rewinds the application to a previously consistent checkpoint [8]. To make the performance results of the two platforms comparable, we disabled checkpointing in Statefun and, in case of delivery error, we proceed in the same way as in Orleans.

Logging. We do not use Orleans storage because we run into the problem of inconsistent state while logging the actor's state [1]. Instead, we use external PostgreSQL to log completed transactions on both platforms. The consistent mechanisms also make the results of the two platforms comparable.

7 EXAMPLE EXPERIMENTS

In this section, we show some example results of our *Online Marketplace* implementations in Orleans and Statefun to investigate the characteristics of our benchmark under different concurrency levels (Section 7.2.1), scaling levels (Section 7.2.2), and workload skewness (Section 7.2.3). The example experiments on scalability are run on a multi-core machine, which is also sufficient for many applications. Extensive experiments on a large cluster need substantial efforts that need to be addressed in future work.

Table 3: Experiment Parameters

Configuration	Parameter	Value
Data Population	Customers	100K
	Products	100K
	Sellers	10K
Transaction Ratio	Update Delivery	60%
	Customer Checkout	30%
	Seller Dashboard	5%
	Update Price	3%
	Delete Product	2%
Customer Checkout	Max Number of Cart Items	10
	Checkout Probability	100%
	Voucher Probability	5%
Distribution	Seller	UNIFORM/ZIPFIAN
	Product	UNIFORM

7.1 Experimental Settings

Deployment. We set up our benchmarking environment on UCloud [26] based on u1-standard instances. A u1-standard contains an Intel Xeon Gold 6130 CPU@2.10 GHz, 32 vCPUs, and 384 GB of memory. UCloud instances run inside a Kubernetes cluster connected via 100Gbps Infiniband virtual network.

To avoid resource competition, we allocate independent instances to different benchmarking components, namely, the benchmark driver, the target platform, and the PostgreSQL database server. To remove cache effects, we restart Orleans and Statefun after each run. In experiments involving PostgreSQL, we also clean up the state after each run. Besides, after data population, we ensure CPU usage returns to idle before initializing transaction submission. We use PostgreSQL 14.5 running in the operating system (OS) Debian 12.1, whereas the driver and platform instances run in the OS Ubuntu 22.10. All the instances are located in the same region and availability zone.

Methodology. All experiments are run in 6 epochs of 10 seconds each with the first 2 epochs as a warm-up period. Our driver measures the two metrics defined in Section 5.1: throughput and end-to-end latency. For each experiment, we maximized the resource allocated to the instances running the benchmark driver and PostgreSQL and we made sure resource usage was kept under 80%, to avoid them becoming the bottleneck.

Workload and Parameters. Table 3 lists the benchmark driver input parameters used in the experiments. We set the probability of payments accepted and checkout to 100% to maximize the amount of transactions. The defined transaction ratio targets mimicking the popular scenario where customer orders contain products from multiple sellers, thus requiring multiple deliveries per order. We use both uniform and skewed distribution for picking sellers per each transaction. The number of sellers and products are picked so to not introduce substantial conflicts in uniform distribution.

Platform Configuration. Statefun introduces a larger configuration space compared to Orleans. To mitigate the possible effects of suboptimal parameters, we systematically experimented many different parameters until we reached the ones that yield the optimal performance under our experimental setup. These parameters can be found in our extended version [15].

7.2 Experiment Results

7.2.1 Effect of Concurrency Level. In this experiment, we refer to *concurrency level* the maximum number of concurrent transactions running in the system at a given time. We measure the overhead incurred by different concurrency levels and identify the parameter that provides the optimal performance in each platform for further evaluation. In this section, we maximize the resources available (32 CPUs) and we set the workload skewness to be uniform for both sellers and products.

As shown in Figure 2(a), Orleans achieves maximum throughput with the concurrency level set to 48. With an even higher concurrency level, more messages would be accumulated on actors' input queue, particularly actors in the checkout critical path (e.g., with *Order* and *Payment* doubling their processing latency).

The overhead introduced by logging is negligible at the start and remains constant as the concurrency level increases, achieving a maximum of 12% overhead compared to non-logging Orleans. The lower throughput is explained by the wait introduced by write requests submitted to PostgreSQL.

On the other hand, in Orleans Transactions, we found that even runs with a lower concurrency level are dominated by lock requests being queued, often leading to message timeouts, thus affecting throughput significantly. Given that this can impact the analysis of the effects of the benchmark and, as Orleans transactions performance degradation under write-heavy scenarios is a known issue [17], we opted not to include it in further experiments.

As for Statefun (Figure 2(b)), in order to achieve maximum throughput, the concurrency level required is significantly higher compared to Orleans. We conjecture several drivers for this: (i) Although we aimed for the most performant deployment style using embedded functions, the containerized deployment mode introduces an overhead on execution functions due to the virtualization layer; [1] (ii) The HTTP ingress acts as a single operator, processing requests serially, introducing a bottleneck; [2] (iii) Although we made sure to adjust the polling rate to a configuration that maximizes Statefun performance, the polling for transaction results coming from the driver in order to compute metrics in a timely manner invariably introduces a processing overhead.

Although negligible at the start, logging shows a steep increase from the 200 concurrency level, maintaining a stable overhead afterward. The difference in overhead from Orleans lies in the benefits of reentrancy [21] found in virtual actors, which minimizes the effect of waiting for responses from external systems.

7.2.2 Scalability. In this experiment, we evaluate the scalability of both platforms by measuring performance metrics as more resources become available. We do so by increasing the number of cores from 4 to 32 and varying the concurrency level accordingly.

Figure 2(c) shows Orleans scales linearly as we increase the number of CPUs. In a similar way, Orleans with logging scales nearly linearly. The effects of logging remain low at the start, slightly increasing from 16 CPUs on and remaining stable afterward, matching the expected overhead found in concurrency experiments (Figure 2(a)).

Statefun can also take advantage of increased computational resources but to a lower degree compared to Orleans. The overhead of logging is nonexistent at the start, and from 16 CPUs on, it impacts Statefun scalability.

Figure 3 shows the latency breakdown of the transactions. In line with the throughput, customer checkout and price update transactions end-to-end latency in Orleans decreases as more CPUs are available. On the other hand, seller dashboard shows increasing latency because the more CPUs available, there are more concurrent transactions in the system, which introduces more customer orders, thus increasing the processing time to compute the query result. Update delivery transaction is not affected by the same phenomena because the processing is independent of the number of orders in progress in the system.

As for Statefun, for customer checkout and update delivery, it is possible to observe latency significantly decreases as more CPUs are available, showing improvements up to 61% from 4 to 16-32.

On the other hand, the overall throughput results are reflected in the longer end-to-end latency. One of the aspects that drives Statefun's overall latency higher compared to Orleans is inherent to the programming model. Whenever an operation involves interacting with multiple functions, a function must send a message to each function and wait for the eventual arrival of the responses. The function must keep track of the responses received through custom-made code, which necessarily involves storing responses in the function's state. On the other hand, in Orleans, multiple actors' calls are encapsulated through promises and do not involve state operations.

A latency breakdown study of the system components affecting Statefun performance is out of the scope of this work. It is worthy noting although Orleans and Statefun offer a platform for programming distributed applications through comparable programming models, they follow disparate architectures, making dissimilarities in performance to spread over many underlying components.

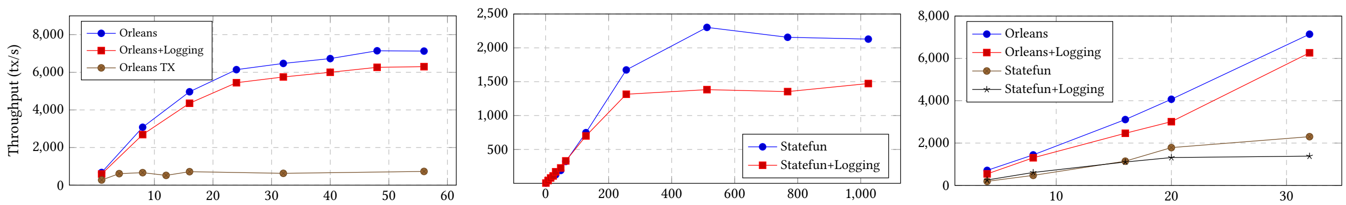
7.2.3 Effect of Workload Skewness. Skew workload causes the access of certain records to become more frequent over time. Therefore, a skewed workload increases contention on a small subset of actors/functions. While we measured the platforms under a low skew level in the previous experiments, in this one, we use the `zipfian` function API in the MathNet library [20] to generate different skewed workloads. To this end, we select different `zipfian` constants to vary the degree of contention in the workload.

In preliminary experiments, we found that fixing seller distribution and varying product skewness showed similar throughput across different skew levels, which led us to investigate the effects of seller skewness. Thus, we picked the sellers using a Zipfian distribution while picking products using a uniform distribution. Again, we use 32 CPUs and the concurrency level that maximizes throughput (Section 7.2.1). Figure 3(c) exhibits the Zipfian value of six skew levels used in the experiments. [3]

[1]We could not confirm this overhead because we do not find instructions in the documentation to run Statefun in bare metal.

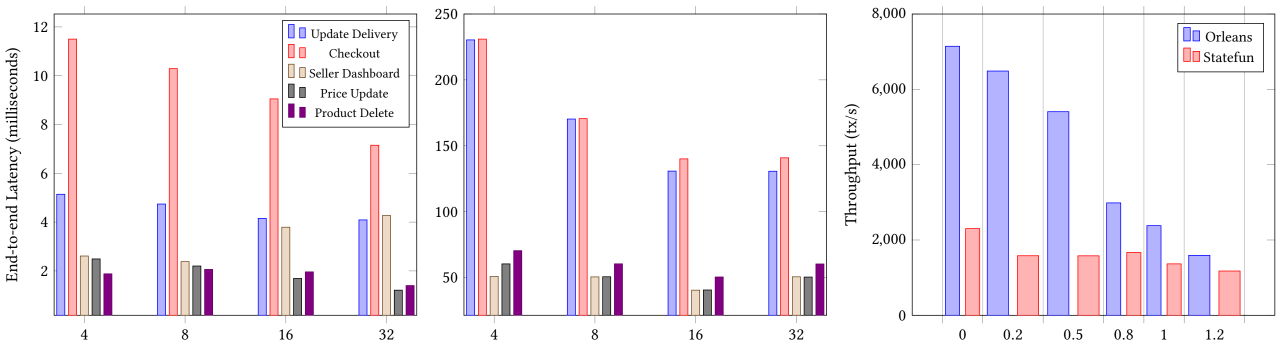
[2]We tried using RichParallelSourceFunction to enable parallel ingress, but we found that driver requests were arriving out of order in functions, impeding the execution of some transactions.

[3]Statefun crashes on skew levels 1 and 1.2. We show the values captured up to the crash.



(a) Concurrency Level (b) Concurrency Level (c) Number of CPUs

Figure 2: Concurrency Overhead (a & b) and Scalability (c)



(a) Number of CPUs on Orleans (b) Number of CPUs on Statefun (c) Skewness

Figure 3: Breakdown Latency (a & b) and Workload Skewness (c)

It is observed that the throughput of Orleans decreases with increasing skewness, following a stable trend. This phenomenon is expected since there is a high contention on certain sellers and their products. Statefun, on the other hand, is less sensitive to workload skewness. In this case, we conjecture that the effects of batching messages play a role, allowing the functions not to incur overhead as skewness increases.

8 RELATED WORK

DeathStar benchmark [10] was created to investigate the effects of microservice architectures on hardware and software in system stacks, particularly network and operating systems. TrainTicket [29] is another benchmark designed to replicate industrial faults in microservices, supporting researchers in fault analysis and debugging. However, while both benchmarks reflect certain aspects of microservice architectures such as functional decomposition and isolation of resources, they fail to address the emerging data management challenges encountered in practice, such as query processing, event processing correctness, data replication, constraints spanning across microservices. Furthermore, they did not clearly specify issues related to transactional guarantees and data invariants, making it difficult to improve data systems for microservice applications.

TPC-W [5] is a transactional benchmark that models the core aspects of user experience on an e-commerce website, such as browsing pages, checking out books, and searching for keywords (e.g., by title). TPC-C [6] was designed to represent the transaction processing requirements of a wholesale supplier. YCSB [4] models transactional workloads in the cloud that are not necessarily executed under ACID semantics. All of them assume a traditional monolithic architecture. Our benchmark models a modern microservice-oriented application and the complex interplay of their components through events, differing substantially from these benchmarks in terms of architectural style and data management requirements.

Unibench [28] offers OLTP and OLAP workloads for multi-model databases. However, it does not model the decomposition of microservices, which results in the absence of distributed and encapsulated states as found in microservice applications. On the other hand, stream processing benchmarks like Linear Road [2] focus on modeling continuous and historical queries, but they do not include transactional workloads.

9 CONCLUSION

Online Marketplace was designed to incorporate data management requirements and challenges faced by microservice practitioners. The benchmark includes an application scenario, a stateful workload generator, and a set of criteria, including transactional guarantees, data replication, event processing, query processing, and constraint enforcement, that can be used to compare competing data platforms.

Through implementing *Online Marketplace* in state-of-the-art platforms for developing distributed stateful applications, we encountered several shortcomings that prevented us from fulfilling the prescribed data management criteria, indicating that *Online Marketplace* can effectively pinpoint the missing core data management features pursued by practitioners. As a result, *Online Marketplace* will support designing futuristic data management systems for microservices.

ACKNOWLEDGMENTS

All of the computation done for this project was performed on the UCloud interactive HPC system.
Dedicated to the memory of José Apolinário Nunes †.

REFERENCES

[1] [n.d.]. *InconsistentStateException with ADO.NET SQL Server grain storage.* https://github.com/dotnet/orleans/issues/4697#issuecomment-398556401 (Accessed on 2023-11-25).

[2] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 480–491.

[3] Sergey Bykov, Alan Geller, Gabriel Kliot, Jim Larus, Ravi Pandya, and Jorgen Thelin. 2010. *Orleans: A Framework for Cloud Computing.* Technical Report MSR-TR-2010-159. https://www.microsoft.com/en-us/research/publication/orleans-a-framework-for-cloud-computing/

[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[5] TPC Council. 2005. *TPC-W: Benchmarking An Ecommerce Solution.* https://www.tpc.org/tpcw/tpc-w_wh.pdf Revision 1.2.

[6] TPC Council. 2010. *TPC Benchmark C revision.* https://www.tpc.org/tpcc Revision 5.11.

[7] Datadog. 2023. *What Is Audit Logging?* https://www.datadoghq.com/knowledge-center/audit-logging/#audit-logs-vs-regular-system-logs (Accessed on 2023-11-20).

[8] Stephan Ewen. 2020. *Stateful Functions 2.0 - An Event-driven Database on Apache Flink.* https://flink.apache.org/2020/04/07/stateful-functions-2.0-an-event-driven-database-on-apache-flink (Accessed on 2023-11-25).

[9] Martin Fowler. [n.d.]. *What do you mean by "Event-Driven"?* Retrieved November, 1st 2023 from https://martinfowler.com/articles/201701-event-driven.html

[10] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* 3–18.

[11] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News* 33, 2 (jun 2002), 51–59. https://doi.org/10.1145/564585.564601

[12] imbursepayments. [n.d.]. *What is a PSP and how does it work?* Retrieved Novemeber, 4 2022 from https://imbursepayments.com/what-is-a-psp

[13] Apache Kafka. 2021. *Kafka Streams.* https://kafka.apache.org/27/documentation/streams (Accessed on 2021-05-02).

[14] Martin Kleppmann, Alastair R. Beresford, and Boerge Svingen. 2019. Online Event Processing. *Commun. ACM* 62, 5 (apr 2019), 43–49. https://doi.org/10.1145/3312527

[15] Rodrigo Laigner, Zhexiang Zhang, Yijian Liu, Leonardo Gomes, and Yongluan Zhou. 2023. *A Benchmark for Data Management Challenges in Microservices.* https://rnlaigner.github.io/files/extended.pdf Manuscript submitted for publication.

[16] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *Proc. VLDB Endow.* 14, 13 (sep 2021), 3348–3361. https://doi.org/10.14778/3484224.3484232

[17] Yijian Liu, Li Su, Vivek Shah, Yongluan Zhou, and Marcos Antonio Vaz Salles. 2022. Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/3514221.3526172

[18] Materialize. [n.d.]. *How Ecommerce Company Drizly uses Materialize for Real-Time Notifications, Alerting, and Personalization.* Retrieved November, 4 2022 from https://materialize.com/customer-stories/drizly

[19] Materialize. [n.d.]. *Materialize: The Operational Data Warehouse.* https://materialize.com (Accessed on 2023-12-02).

[20] Math.NET. 2023. *Math.NET Numerics.* https://numerics.mathdotnet.com/ (Accessed on 2023-11-20).

[21] Microsoft Orleans. 2023. *Request scheduling.* https://learn.microsoft.com/en-us/dotnet/orleans/grains/request-scheduling (Accessed on 2023-11-25).

[22] Apache Flink Statefun. 2023. *Embedded Functions.* https://nightlies.apache.org/flink/flink-statefun-docs-master/docs/concepts/distributed_architecture/#embedded-functions (Accessed on 2023-11-25).

[23] Apache Flink Statefun. 2023. *Shopping Cart Example with Docker Compose.* https://github.com/apache/flink-statefun-playground/tree/main/java/shopping-cart (Accessed on 2023-11-25).

[24] Apache Flink Statefun. 2023. *Stateful Functions: A Platform-Independent Stateful Serverless Stack.* https://nightlies.apache.org/flink/flink-statefun-docs-master/ (Accessed on 2023-11-25).

[25] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. 2018. P-Store: An Elastic Database System with Predictive Provisioning. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 205–219. https://doi.org/10.1145/3183713.3190650

[26] UCloud. [n.d.]. *UCloud User Guide.* https://docs.cloud.sdu.dk (Accessed on 2023-11-28).

[27] Yiwen Wang, Júlio Cesar dos Reis, Kasper Myrtue Borggren, Marcos Antonio Vaz Salles, Claudia Bauzer Medeiros, and Yongluan Zhou. 2019. Modeling and Building IoT Data Platforms with Actor-Oriented Databases. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019.* OpenProceedings.org, 512–523. https://doi.org/10.5441/002/edbt.2019.47

[28] Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. 2019. UniBench: A Benchmark for Multi-model Database Management Systems: Recognizing Outstanding Ph.D. Research. 7–23. https://doi.org/10.1007/978-3-030-11404-6_2

[29] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Trans. Softw. Eng.* 47, 2 (feb 2021), 243–260. https://doi.org/10.1109/TSE.2018.2887384

A APPENDIX

The appendix contains further details about *Online Marketplace* benchmark specification, including the data model (Section A.1), event schema A.2, and the remaining specification of the continuous queries from Section 2.3.2 (Section A.3).

A.1 Data Model

Figure 4 shows the complete data model of *Online Marketplace* microservices.

A.2 Event Schema

Table 4 shows the event schema of *Online Marketplace* microservices.

A.3 Continuous Queries

In this section, we specify the remaining continuous queries presented in Section 2.3.2.

A.3.1 Cart Abandonment. Assuming there is a stream that is the union of both failed payments and failed reservations, one could implement **Cart Abandonment** in ksql [13] through the following continuous query:

```
create stream failed_checkouts as
select cart id and ts
from checkout_cart as c
inner join failed_events on cartid

select failed_checkouts.cartid
from failed_checkouts as fc
left join checkout_cart as c
within 10 minutes ON stream1.cartid = stream2.cartid and stream2.ts
      > stream1.ts
where stream2.cartid is null %and stream1.ts < stream2.ts ts will
      not exist wither...
EMIT CHANGES;
```

A.3.2 Low Stock Warning. The continuous query to warn sellers about low stock can be specified through the following statements in ksql:

```
create stream stock_decrement
payment success
that tells us the stock sold
to decrement from the memory_stock_table

create stream stock_increment to listen from the event
      stock_replenishment
that tells us the stock replenished
to increment to the memory_stock_table

create stream qty_orders_per_week
select productid, ts, sum(quantity) as sum
from checkout_item as c
group by productid
sliding window 1 week

create stream agg_checkout
select productid, sum(quantity)/count(*) as avg
from qty_orders_per_week as q
group by productid
sliding window 1 month

create stream stockthreshold
select productid, avg
from agg_checkout as c
inner join memory_stock_table as mem on productid
where mem.quantity < avg
emit changes;
```

A.4 Statefun Configuration

We followed memory tuning [4] and configuration[5] guidelines in order to configure Statefun instance for our experiments. The configuration used is shown in Listing 4.

```
state.backend=hashmap
jobmanager.memory.process.size=24 gb
taskmanager.memory.process.size=24 gb
taskmanager.memory.managed.size=0 gb
taskmanager.numberOfTaskSlots=6
parallelism.default=6
statefun.async.max-per-task=16000
maxNumBatchRequests: 15
```

Listing 4: Statefun Configuration

[4]https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/deployment/memory/mem_tuning
[5]https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/deployment/config/

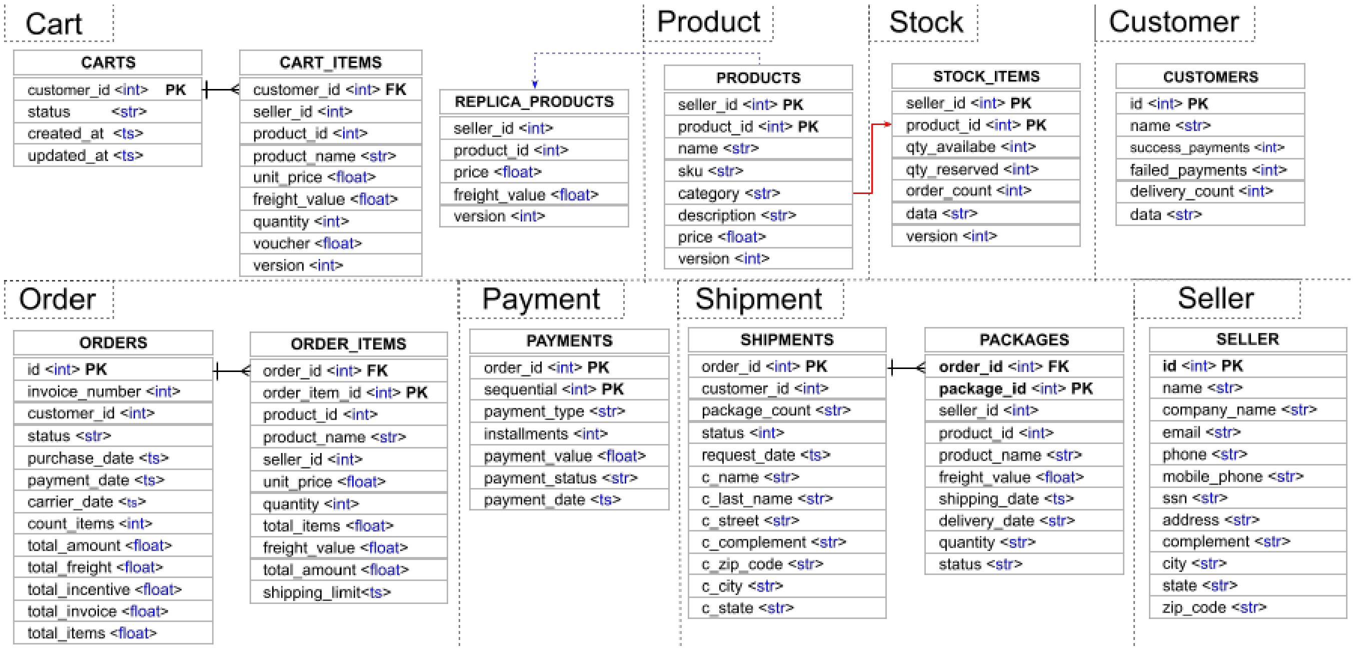


Figure 4: *Online Marketplace* Data model

Table 4: Events Processed by *Online Marketplace*

Event Type	Event Schema	Producer	Consumers	Transaction
`Price Update`	`seller_id, product_id, newPrice, version`	Product	Cart	Update Price
`Product Delete`	`seller_id, product_id, version`	Product	Stock	Delete Product
`Reserve Inventory`	`checkout_info {customer_id, payment_type, card_number, ... }, items: [{seller_id, product_id, price, version}, ... ], timestamp`	Cart	Stock	Customer Checkout
`Stock Confirmed`	Same as `Reserve Inventory`	Stock	Order	Customer Checkout
`Stock Reserva-tion Failed`	Same as `Reserve Inventory`	Stock	Customer	Customer Checkout
`Invoice Issued`	`order_id, invoice_num, total_amount, customer_info {customer_id, payment_type, card_number, ... }, items: [{seller_id, product_id, price, version}, ... ], timestamp`	Order	Payment, Seller	Customer Checkout
`Payment Confirmed`	Same as `InvoiceIssued`	Payment	Order, Seller, Shipment	Customer Checkout
`Payment Failed`	Same as `InvoiceIssued`, but including status	Payment	Order, Seller	Product Update
`Shipment Notifica-tion`	`order_id, invoice_num, shipment_id, shipment_status, timestamp`	Shipment	Order, Seller	Customer Checkout & Update Delivery
`Delivery Notifica-tion`	`order_id, invoice_num, shipment_id, package_id, seller_id, product_id, delivery_status, timestamp`	Shipment	Seller, Customer	Update Delivery