DELFT UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

Benchmarking Stateful Serverless Functions

Author:
Martijn COMANS

Supervisor:
Dr. Asterios KATSIFODIMOS

Co-supervisors:
Dr. Marios FRAGKOULIS
Kyriakos PSARAKIS

A thesis submitted in fulfilment of the requirements
for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

May 3rd, 2023

Declaration of Authorship

I, Martijn COMANS, declare that this thesis titled, "Benchmarking Stateful Serverless Functions" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Martijn Comans

Date: April 27, 2023

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science

Software Technology

Master of Science

Benchmarking Stateful Serverless Functions

by Martijn COMANS

Serverless computing is an increasingly popular paradigm in cloud computing where many of the operational challenges of running cloud applications, like server provisioning and management, are left to the cloud provider. A popular form of serverless computing is Functions-as-a-Service (FaaS), where the user submits functions for which the resources are automatically provisioned and scaled. However, FaaS functions are traditionally stateless, and thus rely on external services to handle state. Stateful FaaS systems are an extension to traditional FaaS offerings which have built-in function state management and function addressability, which allows for functions to communicate and to form complex operations. This work introduces a performance benchmark for stateful functions systems, based on an e-commerce application. The benchmark includes two workflows based on two complex operations that involve multiple stateful functions. The workflows can be dynamically altered to form different function calling structures. We provide reference implementations of the benchmark application for four current stateful FaaS systems, and a benchmark client which runs the two benchmark workloads on these implementations. We show that our benchmark can be used to test and compare stateful systems on performance, networking, cost and scalability, by running various experiments on our reference implementations.

Acknowledgements

Before the start of this work, I would like to thank several people that made this thesis possible. First, I would like to thank Dr. Asterios Katsifodimos for his supervison and feedback throughout this project. Furthermore, I would like to thank Dr. Marios Fragkoulis for his input early in the process. I would also like thank Kyriakos Psarakis for his day-to-day supervision and frequent interesting discussions that shaped this thesis.

Finally, I would like to express my heartfelt gratitude to my friends and family, and in particular Daan, Henk-Jan, Nick, Olav, and Philippe, for their continued support and encouragement during this long process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The emergence of cloud computing brought new opportunities to the IT hardware market (Armbrust et al., 2009). It dramatically reduced upfront costs for developers and businesses to get access to compute resources and allowed companies with large datacenters like Amazon and Google to share and profit from their underutilized resources. In the traditional deployment model, developers needed to provision and operate their own hardware for their applications, whereas in the cloud model, they could now rent computing resources from cloud providers. Since cloud resources are seemingly always available, businesses can scale their resources dynamically according to utilization, reducing costs further.

However, managing cloud applications can still be a complex task for users. Serverless computing is a relatively new paradigm for deploying and running cloud applications, where the developer submits only their code, and the cloud provider is responsible for provisioning and managing the servers on which the code is run (Jonas et al., 2019). In this model, system administration concerns like scaling, fault tolerance and monitoring are left to the cloud provider. Furthermore, the cloud user is only billed for the actual usage of resources. For the cloud provider, it is an opportunity to further optimize the utilization of their hardware (Baldini et al., 2017). One of the most popular serverless models on cloud platforms is Functions-as-a-Service (FaaS), where users submit simple functions that can be triggered to run by various events. FaaS functions are stateless, making them easy to scale and run in parallel. However, this means that functions rely on external services to handle state, like cloud storage or databases. As functions can be executed on any server that is available, data needs to be shipped to wherever the code is running, which can cause increased latency (Hellerstein et al., 2019). Furthermore, functions can not communicate and coordinate with each other directly, as they are short-lived and non-addressable. This makes distributed computing protocols that enable guarantees like data consistency virtually impossible. These limitations make current FaaS offerings not suitable for many stateful, low-latency applications.

Stateful FaaS systems try to address these limitations, by offering built-in function state management and allowing direct function-to-function calls, enabling complex workflows. Examples of Stateful FaaS systems include Apache Stateful Functions[1] and Cloudburst (Sreekanti et al., 2020). As more stateful functions systems are developed and presented, it becomes imperative to test and compare these systems. For example, it is important to test and understand the implications of the various

[1]https://nightlies.apache.org/flink/flink-statefun-docs-stable

design decisions during the development of new stateful functions systems. Furthermore, as these systems become more suitable for production applications, potential users need to be able to evaluate systems on whether they are suitable for their applications. Therefore, we argue that a standardized benchmark is necessary.

Current benchmarks do exist, for database systems (Raab, Kohler, and Shah, 1992; Cooper et al., 2010; Dey et al., 2014) as well as for FaaS systems (Wang et al., 2018; Copik et al., 2021) and microservices (Gan et al., 2019). While some aspects of these benchmarks are relevant for stateful functions systems, they cannot be directly applied. Furthermore, existing benchmarks fail to properly test the features that are unique to stateful functions systems.

This thesis will be focused on the following research questions:

RQ1. How can we design a benchmark to evaluate stateful serverless functions systems?

- RQ1.1 What do current benchmarks for databases and cloud systems offer, that is relevant for testing stateful serverless functions systems?
- RQ1.2 How do we design a benchmark application that uses the unique characteristics of stateful serverless functions?
- RQ1.3 How do we design benchmark workloads and analyses that help us understand stateful serverless functions systems?

RQ2. How do current stateful serverless functions systems compare when tested using this benchmark?

1.1 Contributions

In this work, we propose a performance benchmark designed specifically for stateful FaaS systems. The design of the benchmark is composed of an e-commerce application which is implemented on the tested system and a benchmark client which tests the system by calling operations on the application. The application contains complex operations which run over multiple functions and are unique to stateful FaaS systems. We evaluate the benchmark by implementing the benchmark application on current stateful FaaS systems and using the benchmark client to test and compare the performance of these systems. Furthermore, we show that the benchmark allows us to inspect and compare design decisions within stateful FaaS systems.

1.2 Report outline

In chapter 2, we will summarise related work in benchmarking cloud systems and databases. Next, we describe the design of our benchmark application and client in chapter 3. In chapter 4 we discuss the current stateful functions systems which we have selected to implement our application on. In chapter 5 we describe the implementation details of our benchmark application, for each of the selected systems. After that, we describe the experiments we have performed on current stateful FaaS systems to evaluate our benchmark in chapter 6. In chapter 7, we discuss our work and propose topics for future research. Finally, in chapter 8, we summarise and conclude our work.

Chapter 2

Related work

In this chapter, we will discuss related work on benchmarking stateful cloud systems. No benchmarks currently exist specifically for stateful serverless functions, but existent work on benchmarking other database and cloud systems is relevant. As stateful serverless functions combine the management of application data with running function code, we address benchmarks for database systems and cloud deployment architectures. First, we will address existing benchmarks for databases. Then, we look at benchmarks for serverless functions and microservices.

2.1 Database benchmarks

Many benchmarks for database systems exist, with a prominent example being TPC-C. TPC-C has existed since 1992 and was designed to test the performance of relational database systems (Raab, Kohler, and Shah, 1992). It is centred around an order-entry system and includes several complex transactions related to managing orders, payments and product stock. An even more complex OLTP (online transaction processing) benchmark named TPC-E was approved in 2007 (Transaction Processing Performance Council (TPC), 2015).

However, these benchmarks assume that the tested database systems can provide ACID (atomicity, consistency, isolation, durability) properties, which many cloud systems can not. YCSB is a benchmark introduced by researchers at Yahoo! aimed at testing cloud serving, or "NoSQL", databases, which typically compromise on ACID properties to achieve better scalability and availability (Cooper et al., 2010). The benchmark is focused on two areas: performance and scaling. YCSB evaluates performance by increasing the throughput of requests to the tested system until the system is saturated and the throughput no longer increases, and then measuring the latency of the requests. Scalability is evaluated by adding more machines to the system and measuring how much the latency reduces when the throughput is constant and how much more throughput the system can handle. The workload that YCSB generates is not modelled after a particular application but instead tries to show how various workload characteristics that may appear in any application influence the performance of the tested systems.

Dey et al. (2014) introduced an extension to YCSB called YCSB-T, which adds two benchmark tiers to test transactional overhead and consistency. These tiers were added to evaluate transaction support in emerging NoSQL database designs. In the workloads of these tiers, the requests are transactions containing multiple CRUD (create, read, update or delete) operations. The transactional overhead tier captures the latency of each specific operation and calculates the overhead of running them in a transaction. The consistency tier detects anomalies in the database by running a validation check on each database record after the workload.

2.2 Serverless functions benchmarks

With the introduction of Functions-as-a-Service (FaaS) offerings at most cloud providers, evaluations for these services have started appearing in literature. Most work is focused on evaluating the performance and cost of the various commercial FaaS offerings. For example, Wang et al. (2018) measured the coldstart latency of AWS Lambda, Azure Functions, and Google Cloud Functions. Coldstart latency refers to the extra time it takes the cloud provider to start up a new function instance before the function is run, compared to reusing an existing instance. They further analyzed the scalability of each of these services, measuring how many function instances could be running at the same time. They did these and other experiments to understand better how these services work behind the high-level API that is exposed to the customer.

Copik et al. (2021) presented a collection of benchmark applications and workloads for FaaS platforms, and use them to answer various questions about the performance, cost and inner workings of the most popular FaaS offerings. Their benchmark suite can be extended to include other applications, and to add support for other FaaS platforms.

2.3 Microservices benchmarks

Gan et al. (2019) introduced a benchmark suite of microservices, to test the implications of deploying a microservice application architecture in the cloud. Their suite contains social network, media service, e-commerce, banking and internet-of-things applications. The authors use these applications to evaluate the implications of using microservices on hardware, operating systems, networking, cluster management, and application design. A tracing system is built into the applications to monitor latency in each microservice and to enable precise performance measurements. For example, the authors show that under high load, there is relatively more latency due to networking between services then when compared to latency under low load.

Chapter 3

Design

In this chapter, we will discuss the design of our benchmark. First, we address the limitations of current benchmarks in the context of stateful serverless functions. Then, we describe the design of both our benchmark application and our benchmark client.

3.1 Current benchmarks

In chapter 2, we have shown some existing benchmarks created for database and cloud systems. However, the discussed benchmarks are not well suited to test stateful functions systems, as they are either not easily applicable or do not test the distinctive features of stateful functions. To show this, we first summarize the specifications of the most important existing database benchmarks, TPC-C and YCSB, and then discuss the limitations in the context of stateful functions. Of the types of benchmarks that we discussed in chapter 2, database benchmarks are the most applicable to stateful functions systems since the management of data is the most significant factor in the performance of these systems.

3.1.1 Specifications

TPC-C

The TPC-C benchmark is based on the database model of an order-entry system (Transaction Processing Performance Council (TPC), 2010). The TPC-C specification includes strict criteria for the database systems that it tests. Most importantly, tested systems need to feature ACID transactions. The benchmark schema contains tables for warehouses (Warehouse), districts (District), customers (Customer, History), customer orders (New-Order, Order), order lines (Order-Line), products (Item) and product stock (Stock). The workload of TPC-C consists of several business transactions which execute one or more ACID database transactions. The transactions are comprised of insert, read and update operations on the database tables. The following business transactions are specified:

New-Order Performs a read-update transaction where a user completes a new order. In this database transaction, first information is retrieved from the warehouse, district and customer tables. Then, a new order is inserted in both the New-Order and Order tables. For a random amount of items included in the order, the price is retrieved from the Item table, the stock of that item is retracted from the Stock table, and a new Order-Line row is inserted. Finally, the price is calculated based on the items included in the order and the tax percentages from the warehouse and district.

Payment Performs a read-update transaction where a user completes a payment. In the transaction, the year-to-date balances of the Warehouse and District are updated with the order value. Furthermore, a row in the Customer table is selected either using the primary key, or the last name of the Customer. The balance of this Customer is then updated and a new row is added to the History table.

Order-Status Performs a read-only transaction that requests the status of the most recent order of a customer. First, a row in the Customer table is selected either using the primary key or the customer's last name. Then, the last order corresponding to that Customer is retrieved from the Order table, and all corresponding Order-Line rows are retreived as well.

Delivery Performs a read-update transaction where a batch of up to 10 orders is processed. This batch may be split up into multiple database transactions. For every batched new order in the database transaction that is processed, the oldest row is selected from the New-Order table with a given warehouse and district ID. This row is then deleted, and the corresponding row in the Order table is updated. Corresponding rows in the Order-Line table are updated, and the amount values are summed to calculate the total amount for the order. Then, the corresponding Customer row is selected and updated.

Stock-Level Performs a read-only transaction that checks the stock of recently sold items. For a given warehouse and district, the last order ID is retrieved from the District table. Then, a range query is executed on the Order-Line table to retrieve all items from the previous 20 orders, up to the retrieved last order ID. These items are selected from the Stock table and all rows with a stock amount lower than a given threshold are counted.

The workload of TPC-C is constructed as a mix of the business transactions that are described above. Each of the transactions, except for the New-Order transaction, has a minimum percentage of executions of the total workload mix. The Payment transaction needs to be a minimum of 43% of the mix, and the Order-Status, Delivery and Stock-Level transactions each make up a minimum of 4% of the mix. The measured throughput of the tested system is measured only on the New-Order transaction, as it makes up the rest of the mix of operations.

YCSB

In contrast to TPC-C, the YCSB benchmark is not based on a database model of a particular realistic application (Cooper et al., 2010). Instead, each workload targets just one database table with a variable number of fields. The workloads call one of four implemented operations: insert, update, read and scan. Therefore, to use the benchmark on a new database, only these operations need to be implemented and exposed. Using these operations, low-level workflows can be implemented that apply to many different real-world applications.

The core specification of YCSB includes the following workloads:

Workload A: Update heavy In this workload, 50% of the requests are update operations, and the remaining 50% are read operations. Database keys are chosen using a zipfian random distribution.

Workload B: Read heavy In this workload, 95% of the requests are read operations, and the remaining 5% are update operations. Database keys are chosen using a zipfian random distribution.

Workload C: Read only In this workload, all requests are read operations. Database keys are chosen using a zipfian random distribution.

Workload D: Read latest In this workload, 95% of the requests are read operations, and the remaining 5% of requests insert a new database record. Database keys are chosen using a distribution that favours the records that were inserted the latest.

Workload E: Short ranges In this workload, 95% of requests are scan operations that return up to 100 records per request, scanned in the database from a given start key. The remaining 5% of requests are insert operations. Database keys for the scan operation are chosen based on a zipfian distribution.

3.1.2 Limitations

While the TPC-C application design and workload do include elements that can be used to benchmark stateful functions, the benchmark is mostly not applicable to these systems. Firstly, since it assumes that the underlying database system supports ACID transactions. Many current stateful functions systems do not support any form of ACID database transactions. Of the four systems that we selected to test, which are described in chapter 4, only one offers support for ACID transactions. Furthermore, the TPC-C workload includes operations that are not directly available or suggested in stateful functions systems. For Example, in the Order-Status transaction, rows in the Customer table are selected using a non-primary key attribute, the last name. Stateful functions can be seen as key-value systems, as functions are in principle only addressable by a primary key. Therefore, non-primary key access such as in the Order-Status and Payment transactions is not directly possible. The Stock-Level transaction includes a range query on the Order-Line table to get the latest entries based on the sequential order ID. Again, in distributed stateful functions systems, the keys are not often sequential, and thus such a query is not directly possible.

YCSB is more lenient in its requirements of the tested database. The workloads only access database entries by primary key, and are thus more applicable to stateful functions systems. The insert, update and read operations necessary for workloads A, B, C, and D could be implemented for most stateful functions systems. Therefore, YCSB could be used to evaluate the performance of the data management of single stateful function invocations. However, both YCSB and TPC-C do not test properties that are unique to stateful functions. Stateful functions are distinct from microservices and databases in the way that each specific data entity, or stateful function, is invoked individually, and that these invocations can be scheduled on different machines. Furthermore, an important property of SFaaS systems is that functions, or data entities, can communicate with each other. In a complex operation that involves multiple data entities or functions, the operation can often span multiple machines.

Decisions on stateful functions system design areas such as scheduling can therefore significantly impact the performance of these complex operations. To highlight these changes between systems, a benchmark for stateful functions should have a workload that includes a range of these operations.

3.2 Benchmark application

Like the TPC-C benchmark, we have chosen to base our benchmark on a real-world application to demonstrate realistic performance results. Our benchmark is modelled after a simple e-commerce application. Firstly, because this is a familiar and relatable domain and secondly, since it gives us a framework of data entities and scenarios to construct a workload with a lot of variety in operations. It can be argued that using a real-world application for benchmarking only makes the benchmark relevant for a small set of use cases. However, we try to design the workloads to have different characteristics and to be dynamically adjustable, to make the benchmark results applicable for a broader domain. Our benchmark application is a simplified version of a web store backend, where products are managed and users can submit orders for checkout. The application design consists of data entities, and operations on these entities. In the stateful functions model, each data entity type and its corresponding operations are implemented as a stateful function.

3.2.1 Entities

In the benchmark application, the following data entities are stored and managed:

Product The product entity represents each product in the webstore and keeps track of the stock and price of the product. When products are sold, the stock decreases. Additionally, the product keeps track of other products that are often sold with it.

User The user entity represents a user interacting with the webstore and keeps track of the credits that a user can use to pay for products.

Shopping cart The shopping cart entity represents the shopping cart of a user in the webstore, and keeps track of which and how many products a user adds to it.

Order The order entity is linked to a shopping cart and a user, and keeps track of the status of an order.

3.2.2 Operations

For the **Product**, **User** and **Shopping cart** entities, the application includes simple CRUD (create, read, update, delete) operations. Additionally, the application includes two more complex operations that span multiple data entities, and thus span multiple function instances. These include the checkout operation and the analytics query operation.

Checkout

The checkout operation includes all data entity types and represents the operation that happens when a user wants to check out all the items that are in their shopping cart. First, the contents of the shopping cart need to be collected. Then, for each of the products in the shopping cart, the stock needs to be reduced with the amount of that product in the cart. Additionally, the price of the products needs to be collected and the total for the order needs to be calculated. Then, the user needs to pay that amount from their credits and the order is completed.

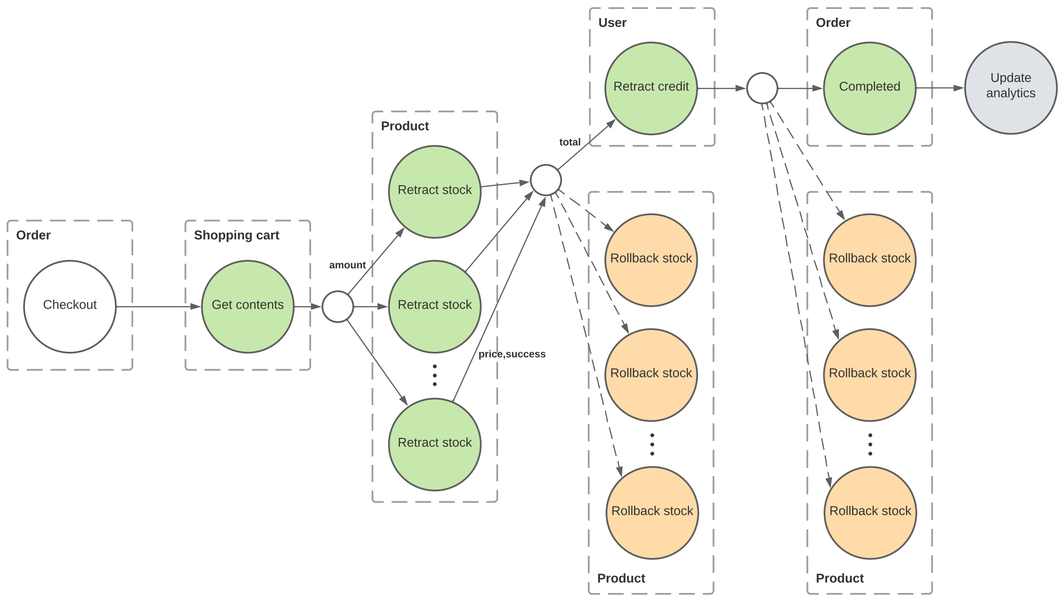


FIGURE 3.1: Checkout operation entity request path

The request path between all the entities in the checkout operation can be seen in Figure 3.1. Each of the nodes represents an operation on a specific data entity, i.e. a function instance, and the borders around the nodes indicate of which entity type they are. As can be seen, after the contents are A rollback mechanism is added, to try to keep the entities in a consistent state. Rollback functions are highlighted in red, and functions part of the successful flow are highlighted in green.

In a successful checkout operation, the order entity calls the shopping cart entity to get its contents. Then, each product in the cart is called in parallel, the stock is retracted and the price of the product is returned. After that, the user entity is called to retract the total for the order from the credits, and the checkout is complete. However, when one or more products in the cart do not have enough stock available, the operation returns unsuccessful. Then all stock changes are rolled back and the checkout fails. Similarly, if the user does not have enough credits, the retract credits operation returns unsuccessful and the stock changes are rolled back. After a successful checkout, all products are updated to include the other products in the order as bought together. This analytics update operation should be run in the background, and the client should not wait for this operation to finish.

The checkout operation is one of the main operations in our benchmark workload, due to two main characteristics of the request structure. Firstly, it shows communication between different types of entities. Some stateful functions systems may handle these types of communications differently, for example because of differences in scheduling protocols. Secondly, the request includes a fan-out calling pattern, when all products included in the order are called concurrently to retract stock. Implicitly, this also includes a fan-in calling pattern, since the operation waits for all retract stock calls to return, before continuing. By including this pattern, we can test how a stateful functions system deals with concurrent calls to different entities within an operation. By varying the number of products that are included in an order, we can inspect the effect of varying amounts of concurrent calls on operation latency.

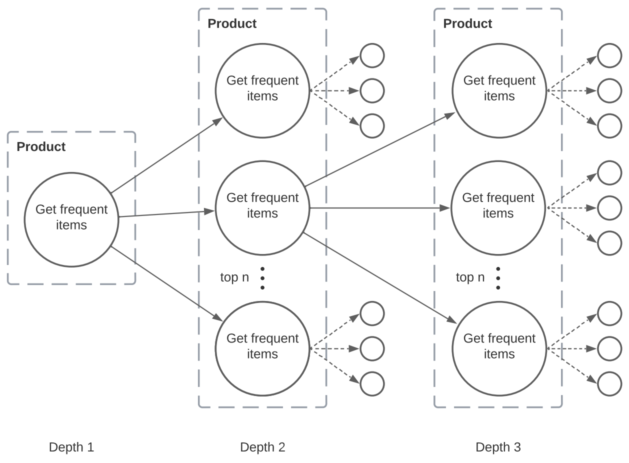


FIGURE 3.2: Analytics query request path

Analytics query

As discussed, each product entity keeps track of which products are frequently bought together with it. The analytics query operation is called on a product to collect this information. Depending on query parameters, it can include recursive calls to other products. Figure 3.2 shows the path of requests an analytics query operation can include. As in Figure 3.1, each node represents an operation on a specific product. First, the "get frequent items" function is called on one product. The call includes two parameters, a top n and a depth d. The top parameter n determines how many of the top products that were frequently bought together with the product that is called should be returned. If the depth parameter d is bigger than 1, each of those top n products is then called with the same operation. This is repeated until the depth d is reached. Since the depth starts at 1, at most $\frac{1-n^d}{1-n}$ nodes are visited during the request. All the collected top products from all visited product entities are then returned from the query.

We include the analytics query in our benchmark workload since it allows us to test the effects of different dynamic calling patterns unique to stateful functions. The calling pattern of the request can be changed using the depth and top parameters. For example, if top $n = 1$ and depth $d > 1$, we get a linear calling structure where each product that is called at most only calls 1 other product. If top $n > 1$ and depth $d = 2$, we get a fan-out pattern, where the first product that is called calls more than one other product. Finally, if top $n > 1$ and depth $d > 2$, we get an *exponential* fan-out pattern, where each product that is called calls multiple other products, as shown in Figure 3.1.

3.3 Benchmark client

The benchmark client is responsible for running benchmark workloads on an implementation of our benchmark application.

3.3.1 Workloads

Two workloads are included in the benchmark, one that is focused on the checkout operation, and one that is focused on the analytics query. These operations are described in subsection 3.2.2.

Checkout workload

The checkout workload simulates the requests of users of the web store application who are buying products. First, the user creates an account, which is represented by the **User** data entity discussed in subsection 3.2.1, and adds credits to this account. They then create a shopping cart and add a uniformly distributed random number of products to their shopping cart. Finally, they run the checkout operation.

Analytics workload

The analytics workload simulates users using the analytics query to request a list of frequently bought together products. The top n and depth d parameters are randomly chosen to generate different calling patterns like described in subsection 3.2.2.

3.3.2 Distributions

In each of the workloads, product entities are randomly chosen to be included in the request. For example in the checkout workload, where the user randomly picks each product that is added to the shopping cart. In the analytics workload, each user randomly picks a product to query from. When running a workload, the random distribution that is used for these random choices can be changed, so the impact of different distributions can be measured. This functionality is also included in the YCSB benchmark (Cooper et al., 2010). We include the following distributions in our client:

1. **Uniform:** We include the uniform distribution as a baseline distribution. This distribution is easiest on the tested system, as the workload is distributed evenly on all products stored in the system. This means that often requests can run concurrently as they do not update the same product.

2. **Zipfian:** The Zipfian distribution simulates that a small number of products are very popular (in the head of the distribution) and a large number of products are not very popular (in the tail of the distribution). This is included since this could give a more realistic simulation of product popularity. This distribution is much more difficult for the tested system, as a small number of products get the most requests, and thus many requests can not be run concurrently.

3.4 Extensibility

The design of our benchmark allows for extensibility in the future. As stateful functions systems are further developed, new properties of these systems may emerge that this benchmark does not sufficiently cover. Therefore, both the benchmark application and the benchmark client can be extended to include more scenarios and tests. In the application, more data entities and operations can be added. The client can be extended to run other workloads and to add tests for other properties such as state consistency. For the existing workloads in the client, more random distributions can be added to change the load on the system.

Chapter 4

Selected systems

In chapter 6, we use our benchmark to test current stateful functions systems. In this chapter, we introduce the systems that we selected for the tests.

4.1 Apache Stateful Functions

Apache Stateful Functions (StateFun)[1] is a stateful serverless application framework that is built on top of the stream processing engine Apache Flink[2]. A StateFun application is composed of addressable functions with local state. Each function can have many instances with different state. Function instances can be called from outside the application, or from other functions. StateFun guarantees fault tolerance for function state.

4.1.1 Architecture

The StateFun architecture is shown in Figure 4.1. The state of a StateFun application is managed by an Apache Flink cluster, as the application functions are transformed into a Flink stream processing application. Each function is represented as a stateful operator in Flink. These operators are interconnected, allowing messaging between them. Apache Flink guarantees that functions and state changes have exactly-once semantics, and fault tolerance through snapshots of state (Carbone et al., 2017). Flink workers, or TaskManagers, listen to incoming events and call corresponding functions. StateFun provides ingress connectors for Apache Kafka and AWS Kinesis. When Apache Kafka is used, the workers listen to specific message topics for each externally invocable function. Each ingress message contains the identifier of the specific function instance that is invoked. For each ingress message, the workers retrieve the state of the invoked function instance, and then call the function implementation, with the saved state. The functions then return any changes in state, which are in return stored by the Flink workers. Functions can be deployed either co-located with the Flink StateFun cluster, or externally as microservices or FaaS functions. This is possible, since the workers call the functions with their state using HTTP or gRPC. Since the state is shipped with every request, the deployed functions themselves are stateless and can easily be scaled. Each function instance can only be called once concurrently, to disallow concurrent changes in state. Functions can send egress messages to Apache Kafka or AWS Kinesis, to communicate results outside of the application.

[1] `https://nightlies.apache.org/flink/flink-statefun-docs-stable/`

[2] `https://flink.apache.org`

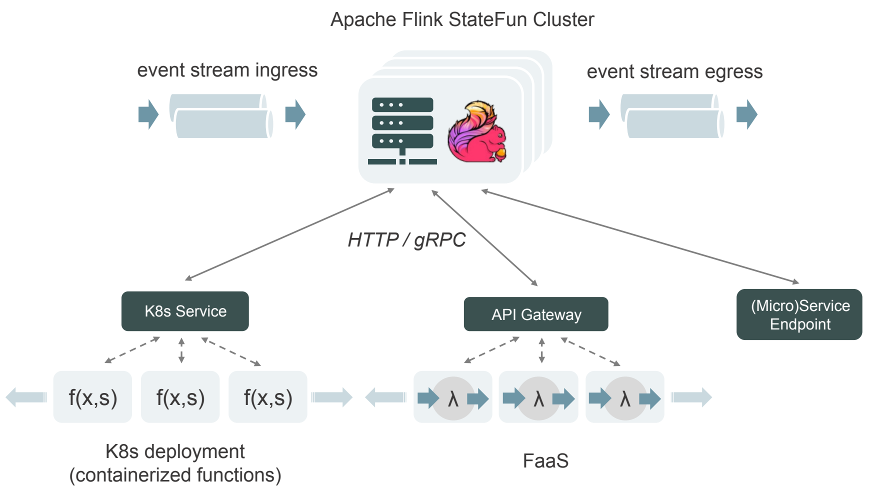


FIGURE 4.1: StateFun architecture, from the StateFun documentation[1].

4.2 Microsoft Orleans

Microsoft Orleans[3] is a framework for developing distributed applications using the virtual actor programming model (Bernstein et al., 2014). Orleans applications are composed of Grains, which are entities with identity, behaviour and state. Grain types are implemented as classes with local state and methods that define its behaviour. Grain types can have multiple individually addressable instances with different state. Grains can be directly invoked from other grains, or from external clients such as API gateways.

4.2.1 Architecture

Grain instances are activated and deactivated on demand by the Orleans runtime. When a grain is invoked while it is not active, the grain state gets loaded into memory. This decreases the read latency of subsequent requests and decreases the load on state storage. When a grain is not used for some time, the grain state is persisted and the grain is removed from memory. Grain instances are hosted on clusters of machines that run so-called silos. Silos coordinate with each other, for the distribution of grains and for fault tolerance. Grains are by default instantiated on a random silo in the cluster. However, this can be changed to other grain placement strategies. When a grain is activated on a silo, its location is stored in a global grain directory. This directory is implemented as a distributed hash table (DHT) over all silos in the cluster. Therefore, when an active grain is called, the silo that is hosting that grain is queried in the DHT, and the call is forwarded to that silo. In the case a silo crashes, invocations on grains that were activated on that silo will be handled by the other silos. Persistent grain state is stored in external storage systems, such as Azure Storage, Amazon DynamoDB, or a relational database management system. Grain methods that change persistent state need to explicitly write the updated state to the external storage system to ensure durability and consistency.

[3]https://dotnet.github.io/orleans/index.html

Transactions

Orleans also offers optional ACID transactions in which grains can participate, which uses two-phase commit (2PC) with a modified locking protocol (Eldeeb and Bernstein, 2016). In many distributed systems that support transactions, two-phase commit with strict two-phase locking (2PL) is used to guarantee atomicity and isolation for the transaction. With strict 2PL, write locks on data included in the transaction can only be released after the transaction is completed. In Orleans, locks on grain state are released immediately after the prepare message in the 2PC protocol, so that other transactions can write changes to that grain. However, subsequent transactions that include the same grain are marked as dependent, while the first transaction is not committed yet. The transaction coordinator keeps a list of all dependent transactions. If any transaction aborts, all dependent transactions abort as well. In this way, the transaction throughput on popular grains is increased while isolation is still guaranteed.

4.3 Cloudstate

Cloudstate[4] is a stateful serverless platform based on the Akka[5] virtual actor framework. Cloudstate applications are composed of addressable data entities with state and behaviour, or stateful functions.

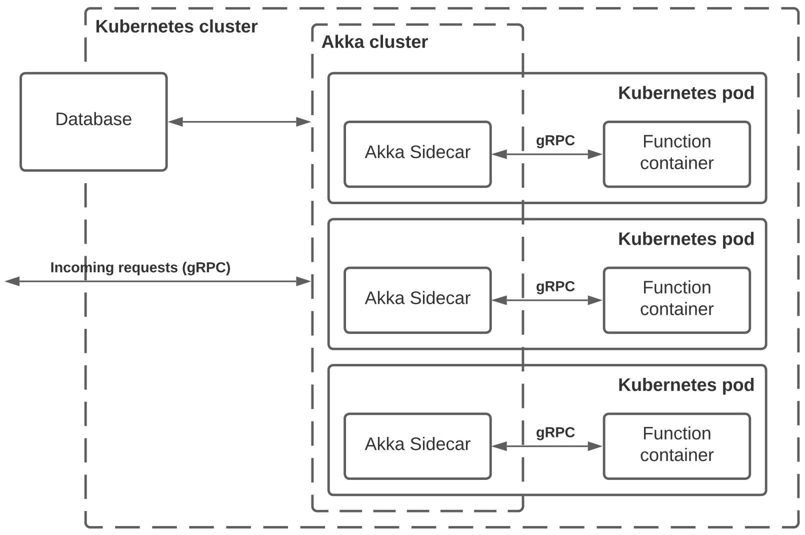


FIGURE 4.2: Cloudstate architecture diagram, based on the cloudstate documentation[4].

4.3.1 Architecture

In Figure 4.2, the general architecture of Cloudstate is shown. In Cloudstate, user-defined functions are deployed as containers on a Kubernetes cluster, while the state of these functions is managed by Akka sidecars. These sidecars are automatically deployed by Cloudstate for each function container, and form an Akka cluster of

[4]https://cloudstate.io/docs
[5]https://akka.io

stateful actors. For fault tolerance, the stateful Akka actors are persisted on an external distributed datastore. The sidecars are deployed in the same Kubernetes pod as their corresponding function container, and host actors that keep the state of the corresponding entities in memory. Since function execution and the corresponding state is located on the same machine, a lower latency is expected. Functions, or entity types, can have multiple instances with state, each addressable via a unique key. Entities can have two types: event sourced and conflict-free replciated data type (CRDT). Event sourced entities are persisted as a journal of change events. In case the actor storing a event sourced entity crashes, the state of the entity can be reconstructed by replaying the journal of events. Periodically, event sourced entities store snapshots to decrease the time needed to reconstruct an entity. Event sourced entities only live on one machine at a time, and cannot be updated by two requests concurrently. CRDT entities are based on mathematical data structures that can be replicated over multiple machines, can be concurrently modified on these machines without coordination, and can eventually be merged without conflicts, to get the actual state of the data type. If there are multiple containers deployed for the same function type, function instances are distributed between them. All incoming and outgoing communication with the functions goes through the sidecars. The sidecars communicate with their user-defined function via gRPC. Therefore, all languages with gRPC support can be used to implement the function behaviour. External communication to the cluster, e.g. from a gateway, is also done via gRPC.

4.4 Cloudburst

Cloudburst (Sreekanti et al., 2020) is a stateful serverless functions framework based on the Anna key-value store (Wu et al., 2019; Wu, Sreekanti, and Hellerstein, 2019). Anna is an autoscaling key-value store (KVS) that stores state in mergeable monotonic lattice data structures, a type of conflict-free replicated data type (CRDT). Serverless functions registered to the Cloudburst runtime can read and write state stored in Anna. All functions have access to any state that is stored in Anna by key, so there is a clear separation between function code and state. This makes Cloudburst different from the other discussed systems, where each function or entity can access only its own state.

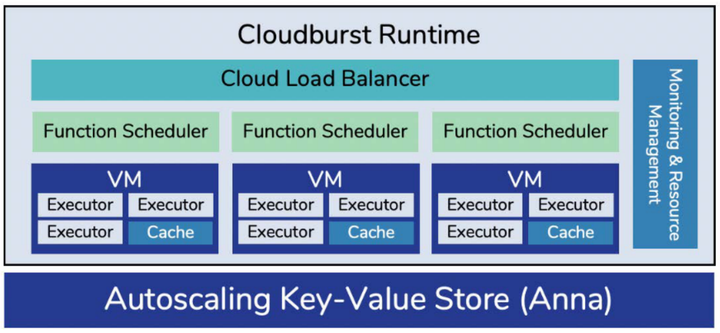


FIGURE 4.3: Cloudburst architecture diagram (Sreekanti et al., 2020)

4.4.1 Architecture

The architecture of the cloudburst system is shown in Figure 4.3. The user-registered functions are scheduled on a cluster of virtual machines, which host function executors. The function code itself is stored in the Anna KVS after it is registered by the user. When a function request comes in, the request is forwarded by the load balancer and scheduler to a function executor. The function executor then retrieves the function from the database and invokes it. If any state is accessed in the function, the executor retrieves that state from the KVS during execution. Each VM that is running function executors keeps a cache of data from the KVS that is accessed by the functions, to achieve lower latency. Executors read persistent state through the cache, and the KVS is only queried if the requested key is not present in the cache. The caches periodically push a list of their cached keys to the KVS, which in turn periodically pushes updates to the caches that have outdated values. All state stored by functions is wrapped in lattice data types by the function runtime. As all functions can read any state from the KVS by key, the same value can be read and updated concurrently. By default, values are stored in a lattice that merges using a last-writer-wins (LWW) strategy. So, if a key-value pair is changed on multiple machines concurrently, the last written value is kept when the state is merged to keep it consistent on all replicas. However, this can lead to loss of data. Workflows of multiple functions can be composed and registered as directed acyclic graphs (DAGs), which forward the result of one function to the next when executed. When a DAG is executed, Cloudburst guarantees repeatable read consistency within the DAG. This means that if one function in the DAG reads a certain key-value pair that was read by a previous function in the same DAG execution, it either sees the last update from within the DAG, or the same value that was read earlier. Cloudburst DAGs cannot be seen as transactions, as there are no atomicity and isolation guarantees.

4.5 Comparison

While the systems described above all provide some form of stateful serverless functions, the architectures of these systems differ significantly.

4.5.1 State management

One of the most relevant differences for our experiments is how the systems manage function state, since this has a large impact on request throughput and latency. In Table 4.1, the main differences in state management are shown. We specifically show where the state of a function is located with respect to where the function is executed, and if there is a cache of the function state local to the function execution. This impacts if and how state is transmitted inside the system cluster, which in turn impacts the latency of a request due to networking. Therefore, we also describe how state is transmitted during a request.

In all systems, the state is persisted externally to the execution of the functions. In StateFun however, it is possible to execute the functions natively on the Flink cluster where the state is managed. However, this brings significant disadvantages, and therefore it is recommended to deploy the functions separately. All systems except StateFun include some form of caching of function state. In Orleans, if a Grain is active, its state lives in the memory of the Silo that executes the grain. In Cloudstate, the function state is present in the memory of the sidecar co-located on the

TABLE 4.1: Comparison of state management in selected stateful functions systems.

System	Persistent state location	State cache location	State transmission
StateFun	External (or co-located) Flink cluster	Not applicable	State transmitted to function execution for every invocation from Flink cluster. After the invocation, the state is transmitted back to the Flink cluster.
Orleans	External database	In function execution memory	State transmitted to external database for every invocation that updates state, state transmitted from database when function (Grain) is not active.
Cloudstate	External database	Co-located on machine with function execution	State transmitted from sidecar co-located on the same machine with function execution for every invocation. If state is updated, a change event or a CRDT is transmitted to the external state backend.
Cloudburst	External database	Co-located on machine with function execution	State is retrieved from database if the cache on the function executor machine does not contain the requested key. Otherwise, the state is only retrieved from the machine-local cache. If updated, the state is written to the database in the background.

same machine as the function execution. In Cloudburst, key-value pairs are cached on the machines that have function executors. It is expected that the systems that implement caching have lower latency when the state of a function instance is read by multiple subsequent requests.

Chapter 5

Implementation

In order to evaluate our benchmark design, we implemented the benchmark application, discussed in section 3.2, in all selected systems that are discussed in chapter 4. In this chapter, we first discuss the details that all implementations share. Then, we discuss the details of each individual implementation. Furthermore, we discuss the implementation of the benchmark client.

5.1 Shared

5.1.1 API

In order to make the benchmark client usable for all current and future application implementations, all implementations must share the same interface. We have chosen to make this an HTTP interface since HTTP APIs are widely used and tools to stress test these APIs are available. The API schema can be found in Appendix A.

5.1.2 Tracing

To evaluate the performance of the tested systems, we measure the latency and throughput of requests fired at the benchmark application. For complex requests that visit multiple functions, like the checkout request specified in subsection 3.2.2, we would like to see how a request progresses through the system and measure specific latency per function. This would allow us to get a better idea of the impact of networking between function instances on latency. To enable the tracing of a request, we assign an ID to each external request which comes into the system. This ID is then passed in each successive internal request through the system. For each function, when an (internal or external) request comes in, the start and end of the function execution are logged. Using the timestamps of the logs, we can then calculate the internal latencies of networking and serialization, and function execution.

5.2 Apache Stateful Functions

In Apache Stateful Functions (StateFun), functions are called from outside the application via Apache Kafka, a message broker. As we want all benchmark applications to share an HTTP API, an API gateway layer is added which forwards all incoming HTTP requests to the Kafka ingress. In Figure 5.1 the basic architecture of the application is shown. The function code is deployed separately from the StateFun cluster and called via HTTP.

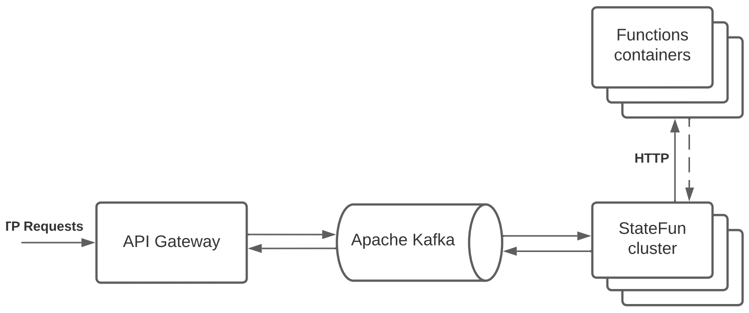


FIGURE 5.1: Apache Stateful Functions application architecture

5.2.1 Functions and Messages

StateFun includes software development kits (SDKs) for Java, JavaScript, Python, and Golang. We have chosen to implement the application functions using the Java SDK since it was documented well. We have implemented the functions in the Kotlin programming language, which fully interoperates with Java and runs on the Java Virtual Machine. For each data entity described in subsection 3.2.1, we created a function class, which defines the function state and logic. Based on which message is passed to the function, either externally via Kafka or internally, different operations on the state are executed.

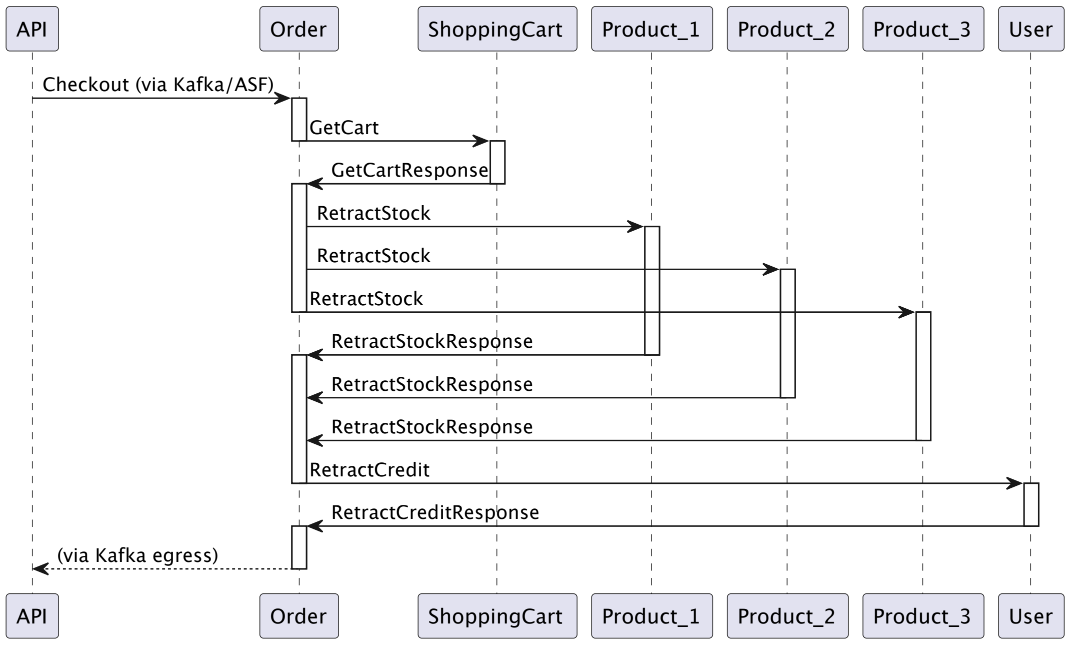


FIGURE 5.2: Sequence diagram of checkout operation in the Apache Stateful Functions implementation

The functions are accessible for the StateFun cluster to call via an HTTP interface, which is handled by the StateFun SDK. The web server which is recommended in the StateFun docs, Undertow[1], is used to host the functions. Communication to and between functions in StateFun is done asynchronously, by sending messages to

[1] `https://undertow.io`

specific functions. An example of complex communication between multiple functions can be found in Figure 5.2, where a sequence diagram of the checkout operation is shown. In this diagram, the messages that are sent between functions are shown. First, the `Checkout` message is sent to the order function, which orchestrates the checkout operation. The order function then sends a message to the shopping cart to retrieve its contents, and stops execution. When the shopping cart function then sends a reply back to the order function, it resumes execution and sends `RetrackStock` messages to all products in the shopping cart. Again, after all messages are sent, the order stops execution. When each response from the products is received, the order function saves the response in its state and stops. When all responses are received it continues and sends a `RetractCredit` message to the corresponding user function. Finally, when the response from the user is received, the checkout is complete.

The analytics query, which was discussed in subsection 3.2.2 is implemented in a similar way. Each product sends messages to its top products and keeps track of which of them have replied. When all other products have replied, the product itself replies to its caller.

To enable tracing as specified in subsection 5.1.2, all messages are wrapped in a wrapper message which contains a request identifier. This identifier is then included in all logs relating to that request.

5.2.2 Gateway

The API Gateway is implemented in the Kotlin programming language, using the Spring Boot and Spring Web frameworks[2]. The gateway conforms to the shared API schema and sends messages to StateFun functions via Kafka that correspond to the API actions. Since StateFun works asynchronously, each HTTP request to the gateway is closed once the message is sent. To get a proper indication of full request latency, the gateway logs when a request comes in, and also when the full request is done. The gateway knows when to log the end of a request by listening to an egress Kafka topic, where StateFun functions submit messages to whenever the full request is done. This can also be seen in Figure 5.2, where at the very bottom an egress message is sent by the order function to the gateway via Kafka.

5.3 Microsoft Orleans

Microsoft Orleans only provides a C# SDK, so the benchmark application is implemented in the C# language. All data entity types are implemented as grain classes, which include their state and functions with logic. Communication between grains is done automatically via Orleans, as interface functions of a specific grain can directly be called from other grain code, or from another colocated .NET process, such as an API gateway. The basic architecture is shown in Figure 5.3. The API Gateway, which is built using the API.NET web framework, is deployed and run together with a grain silo, which starts, runs, and persists the implemented grains. For the persistence of grain state, Orleans offers connectors to cloud databases like Azure CosmosDB or AWS DynamoDB, or to a relational database. We have chosen to use the relational database connector, since it allows us to pick an open source database system like PostgreSQL.

[2] `https://spring.io`

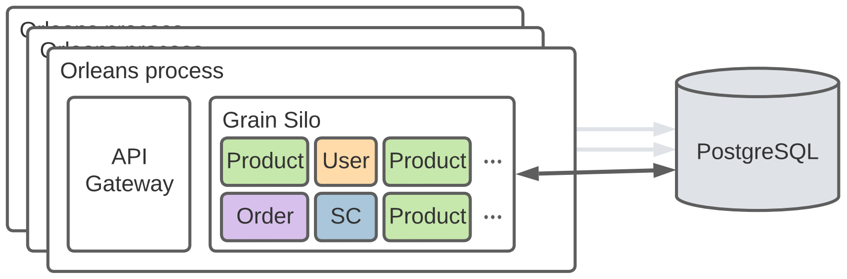


FIGURE 5.3: Microsoft Orleans application architecture

Orleans uses the async/await functionality of C# when calling functions of Grains, allowing the caller to wait for the asynchronous function to return. Using this functionality, the API gateway waits for a grain call to return before returning the HTTP call. It is also used to call multiple grains concurrently and wait until all return when there is a fan-out pattern, such as in the checkout operation where the order grain calls multiple products (see Figure 3.1).

For the tracing of complex operations, we make use of the request context feature of Orleans. This enables us to set and store a request identifier when the operation is called from the API gateway, which will then automatically be passed in any subsequent requests to other grains, where it will be available to include in the logs. As the API gateway waits for the asynchronous grain call to finish, it can log both the start and the end of the request, so end-to-end latency can later be calculated.

During the testing of the analytics query, we observed that deadlocks could occur in certain queries. Orleans by default only allows grain instances to execute one operation at the same time. In the analytics query, product grains call other product grains asynchronously but wait for the results of that call. Since cycles are possible in the analytics query call pattern, this could lead to deadlocks. However, Orleans includes functionality to interleave grain calls, allowing later calls to execute while one grain call is waiting for an asynchronous operation. When we annotated the query methods to allow interleaving, no deadlocks were present anymore.

5.3.1 Transactions

As discussed in section 4.2, Orleans offers optional ACID transactions for changes to grain state. Our base Orleans implementation does not use ACID transactions since the other selected systems do not offer the same functionality. However, in our experiments, we also wanted to test a transactional Orleans implementation to show how much overhead Orleans transactions add. Most of our transactional implementation is copied from the base implementation, but the grains were adapted to use the transactional state API. Furthermore, all grain methods needed to be allowed to interleave, as otherwise, deadlocks could occur due to transaction protocol operations.

5.4 Cloudstate

The Cloudstate implementation of the benchmark application was written in the Kotlin programming language. Cloudstate provides SDKs in various languages, but we chose a JVM-based language like Kotlin since the Java SDK was one of the most well-documented. Each of the data entity types was implemented as an event-sourced entity in Cloudstate. In such a data entity class, each mutating operation generates an event to mutate the state of the data entity. Operations on data entities are called using the gRPC protocol[3]. An API gateway was added to conform to the common API specification. This gateway is very similar to the gateway that we've implemented for the StateFun implementation, as it is also built using Spring Boot and Spring Web and implemented in Kotlin. The gateway for Cloudstate converts the incoming HTTP request into gRPC calls to the Cloudstate entities. As gRPC calls are synchronous, the gateway waits for the function calls to finish before returning the HTTP request. As in the Orleans gateway, the start and end of the HTTP request are logged to enable calculating the request latency later. In complex requests where entities call other entities, gRPC is also used. Tracing was implemented by adding a request identifier to each gRPC payload and manually passing it to each subsequent call in a complex request.

5.5 Cloudburst

The Cloudburst application is written in Python, since it is the only language that Cloudburst provides an API for. Cloudburst does not provide a paradigm for writing data entities or specifically addressable functions with state. Instead, Cloudburst functions can access a key-value store to access state. In the application implementation, each operation on a data entity is written as a separate Cloudburst function. Functions that mutate the state of an entity first retrieve state from the key-value store, then mutate it and finally write it back to the store. The API gateway is also implemented in Python, using the Flask framework[4]. It uses the Cloudburst API to call the functions that correspond to the requested route, with the provided data entity identifiers. Cloudburst calls are synchronous, and so the gateway waits for the function to finish before returning the HTTP request.

To construct complex operations that use multiple functions, Cloudburst provides an API to construct directed acyclic graphs (DAGs) of functions, which can then be called as a single operation. However, this functionality is not sufficient to construct a complex call with a fan-out pattern, such as the checkout operation shown in Figure 3.1. Firstly, because there is no functionality to construct DAGs with a dynamic number of function calls, which is necessary in the checkout operation since the number of products in a shopping cart can differ, and thus a dynamic number of products are called. Furthermore, if a static DAG with a fan-out call is registered, each of the functions that are concurrently called always receives exactly the same parameters making it impossible to distribute work over these functions.

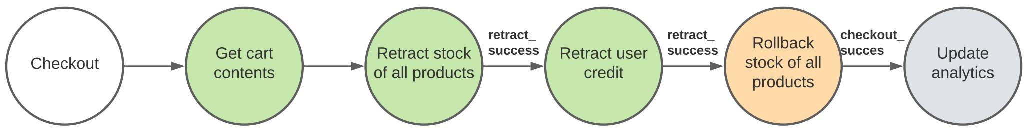


FIGURE 5.4: Adjusted checkout operation request path in Cloudburst implementation.

[3]https://grpc.io
[4]https://flask.palletsprojects.com

With these limitations on DAG structure, we could only construct complex operations with no fan-out patterns. For example, we implemented the checkout operation with only a linear chain of calls. The calling structure of the checkout implementation is shown in Figure 5.4. After the contents of the cart are retrieved, a single function retracts the stock of all the products in the cart. Since no dynamic changes to the DAG can be made, it is also not possible to abort the checkout early when for example not enough stock is available. We implemented the rollback mechanism by including a single function in the DAG after all the checkout steps are completed. If there is not enough stock for one or more of the products in the shopping cart, a failure flag is passed through all subsequent calls. If the failure flag is set, no credit is retracted from the user, and only after that the stock of all products is rolled back to its initial state. If there is enough product stock but the user does not have enough credit, the failure flag is again set, and the products are rolled back in the last step. The rollback step is always executed, but ignores the rollback if the checkout was successful.

5.6 Client

The benchmark client is implemented in Python, and has two main responsibilities. First, it generates the benchmark workloads used to test the benchmark application implementations. Furthermore, it collects the logs that the benchmark application produces to later analyze the performance of the tested system. The basic architecture of the client is shown in Figure 5.5 and explained below.

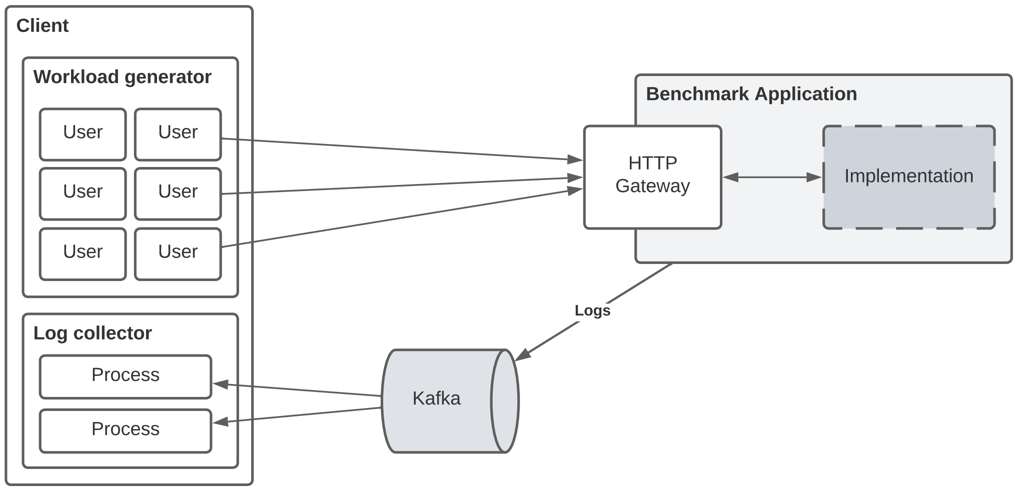


FIGURE 5.5: Client architecture

5.6.1 Workload generation

To generate benchmark workloads, the client uses the Locust[5] library to simulate a large number of users which send requests to the HTTP API of the application. The two workloads discussed in subsection 3.3.1 are implemented as Locust user classes. Before the checkout workload is run, the client generates a specified number of products in the tested system, which can then be used in the workload. The

[5]https://locust.io

simulated users then choose random products from the generated product set to add to their shopping cart using a specified random distribution, as discussed in subsection 3.3.1. The identifiers of the generated products are saved, so they can be reused in other benchmark runs.

5.6.2 Log collection

The client assumes that all logs that are generated by the benchmark application are sent to an Apache Kafka topic. All of the selected systems are deployed on Kubernetes, and so we run a fluent-bit[6] process on each Kubernetes node to forward all logs to Kafka. The benchmark client spawns one or more Python processes that act as Kafka consumers and save all received logs locally so that they can be analyzed later.

[6] https://www.fluentbit.io

Chapter 6

Experiments and evaluation

In order to evaluate our benchmark, we run experiments on the implementations of our benchmark application, using our benchmark client. We show that using our benchmark, we are able to test and compare these systems in three domains: performance, networking and scalability. In this chapter, we first discuss the general setup of our experiments. After that, we describe the experiments and show and discuss the corresponding results.

6.1 General setup

For all experiments, we use Google Cloud[1] resources. As all of the selected systems run on Kubernetes clusters, we use Google Kubernetes Engine (GKE) to provision and manage the test clusters. For all deployments, Kubernetes version 1.23 is used. Unless specified otherwise, we use clusters of `n2-standard-4` machines (4 vCPUs, 16GB memory), with 100GB SSD storage. These nodes run the default container-optimised operating system provided by GKE. We use the managed Prometheus service[2], to collect resource usage metrics exported from the Kubernetes cluster.

Unfortunately, our Cloudburst implementation is missing from any of the large-scale experiments in this chapter, as we were unable to deploy Cloudburst and its corresponding stack of services on GKE. The Hydro cluster repository[3] does include instructions for deploying the stack on AWS EC2 machines using a custom script, but this script does not work for deploying to other cloud providers.

6.1.1 Workloads

For our experiments, the workloads described in section 3.3 are used. As described in subsection 3.3.2, either a uniform or zipfian distribution is used to select which product IDs are used in the benchmark. In the experiments described below where a zipfian distribution is mentioned, we use a zipfian distribution parameter of 1.25. In our checkout workload experiments, the simulated user picks a uniformly distributed random number between 2 and 8, and adds that number of products to its cart. The simulated users only choose from a list of 5000 products. In the experiments using the analytics workload, the simulated user picks randomly from three options relating to the analytics query parameters described in subsection 3.2.2:

[1] `https://cloud.google.com`
[2] `https://cloud.google.com/stackdriver/docs/managed-prometheus`
[3] `https://github.com/hydro-project/cluster`

1. A query with a linear calling pattern is requested: A top parameter of $n = 1$ is chosen and the depth parameter is randomly chosen between $1 \leq d \leq 20$.
2. A query with a fan-out calling pattern is requested: A depth parameter of $d = 1$ is chosen and the top parameter is randomly chosen between $2 \leq n \leq 20$.
3. A query with an exponential calling pattern is requested: The top parameter is randomly chosen between $2 \leq n \leq 10$ and the depth parameter is then randomly chosen between $2 \leq d \leq \lfloor \log_n 1000 \rfloor$.

6.2 Performance

Our benchmark is focused on testing the performance of stateful functions systems. The two main metrics of performance that we measure to compare the systems are request throughput and request latency. First, we test the maximum throughput that the systems can handle when running our workloads. We then use the measured maximum throughput and run the systems at a fraction of this throughput to measure latency under a stable load. These experiments are described below. We test the performance of the selected systems when deployed on a cluster of 5 machines. On this cluster, 1 machine is used only to run an instance of Apache Kafka, which handles the export of the application logs. In the deployment of the StateFun application, this Kafka instance also handles the communication to the StateFun cluster. The other 4 nodes are used to deploy containers for the application.

6.2.1 Maximum throughput

Experiment question

What is the maximum throughput that the stateful functions systems can handle when running the checkout and analytics workloads, and what is the influence of the random distribution used in the workload on this throughput?

Experiment design

We measure the maximum throughput that each 5-node setup can handle by increasing the amount of simulated users in steps, each time letting the system acclimate to a higher number of requests per second. We define the maximum throughput as the maximum of the 10-second moving average of the measured throughput. We take the maximum of the moving average instead of the actual measured maximum throughput, as due to batching or other scheduling mechanisms in some seconds many more requests finish than in others, which gives a false indication of the useful throughput of the system. In the checkout workload, a locust user performs multiple requests per simulation before finishing the checkout, but we measure only the throughput of the final checkout request. For the analytics workload, only one request is sent per simulation, which is used to measure the throughput.

An example can be found in Figure 6.1, where a maximum throughput experiment of the Microsoft Orleans implementation is shown. In this case, the checkout workload is used, and a uniform distribution is used to select products. In the figure it can be seen that during the workload, throughput is increased in 2-minute steps. The throughput of the system, or more specifically the amount of requests that are finished each second is shown in blue. The 10-second moving average is shown

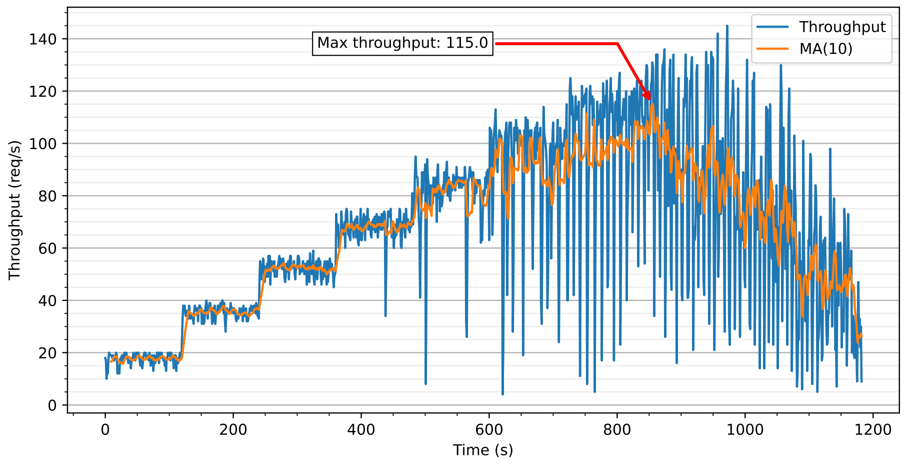


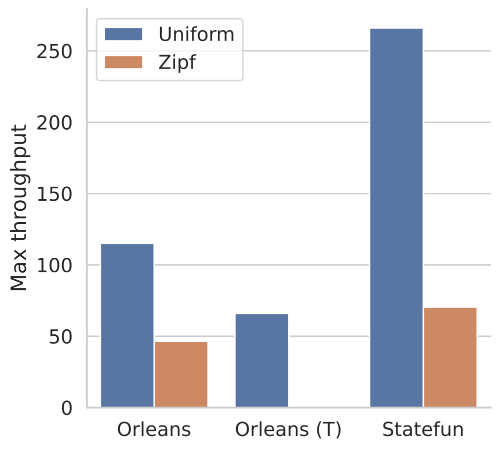
FIGURE 6.1: Throughput of checkout requests during a checkout benchmark workload, for the base deployment of the Orleans benchmark application.

in orange and its maximum value, which we use as maximum throughput, is annotated. In the maximum throughput experiments, the benchmark is run for 20 minutes, and the throughput is increased in 2-minute intervals.

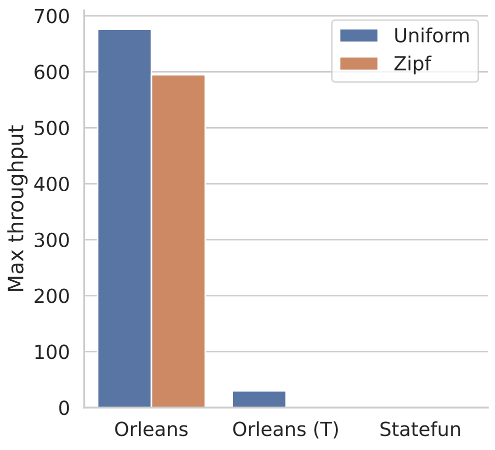
Results

In Figure 6.2 the results of our maximum throughput experiments are shown. In blue, the maximum throughput while using a uniform distribution for product selection is shown, while in orange the result of using a zipfian distribution is shown. In Figure 6.2a we see the maximum throughput for the checkout benchmark. In both the Orleans and StateFun implementations there is a clear drop in throughput when the zipfian distribution is used, compared to the uniform distribution. However, the drop in throughput is much higher for the StateFun implementation when compared to Orleans. With StateFun, we observe a 73.5% drop in throughput when a zipfian distribution is used, while with Orleans this drop is 59.5%. Furthermore, we can see that when a uniform distribution is used, StateFun can reach a maximum throughput that is more than twice as high as Orleans. When comparing the results of non-transactional Orleans with transactional Orleans, we see that the transactional overhead is significant. When a uniform distribution is used, transactional Orleans has a throughput that is 42.6% lower than the non-transactional version. Furthermore, the transactional Orleans implementation cannot sustain any reasonable throughput when a zipfian distribution is used.

In Figure 6.2b, the maximum throughput results are shown for the analytics benchmark. For both the uniform and zipfian distributions, StateFun could not sustain any throughput and the StateFun runtime crashed often while running the benchmarks. The transactional Orleans implementation could not sustain any throughput while using the zipfian distribution. In contrast, the non-transactional Orleans implementation is able to reach a maximum throughput that is much higher than the checkout workload throughput. For Orleans, we can see that the drop in throughput when using a zipfian distribution is less severe than in the checkout benchmark, 12.0% compared to 59.5%. This can be expected, since the analytics data that is queried,



(A) Checkout benchmark



(B) Analytics benchmark

FIGURE 6.2: Measured maximum throughput in requests per second of the tested systems while running the checkout and analytics workloads. The Orleans implementation which uses ACID transactions is labelled as "Orleans (T)". Missing results signify that no sustainable throughput could be reached.

is populated using a uniformly random selection of products, and thus the effect of using a zipfian distribution is less severe. The transactional Orleans implementation is able to sustain some throughput, although it is only able to reach a maximum throughput of 29.9 req/s, 95.6% less than the non-transactional implementation.

6.2.2 Request latency

In order to compare the time that a client would have to wait for a request to finish, we measure the request latency when the systems are running under a stable load. As explained in chapter 3, the workloads are designed to have varying characteristics in the request structure, to highlight how the systems handle concurrent and subsequent calls within an operation. In this experiment, we therefore also measure the impact of these varying characteristics of a request on its latency.

Experiment questions

1. What is the mean latency of a request when the system is running the benchmark workloads under a stable load, and how does the random distribution used in the workload impact the latency?
2. How does the amount of products included in an order influence the latency of the checkout request?
3. How do the analytics request parameters influence the request latency?

Experiment design

We define a stable load as 80% of the maximum throughput tested in subsection 6.2.1. As our benchmark uses simulated locust users, we can not simulate a specific throughput with great accuracy. Simulated users wait 1 second between running their tasks,

TABLE 6.1: Latency numbers of checkout workload at stable throughput.

System	Distribution	Mean latency (ms)	95%	99%	Mean throughput (req/s)	Errors
Orleans	Uniform	14.2	24.2	116.8	80.5	0.08%
Orleans	Zipf	59.2	240.1	693.5	31.2	0.0%
Orleans (T)	Uniform	24.8	33.9	71.9	46.3	0.51%
StateFun	Uniform	256.6	413.0	968.3	164.4	16.42%
StateFun	Zipf	266.7	427.0	503.0	51.6	4.06%

TABLE 6.2: Latency numbers of analytics workload at stable throughput.

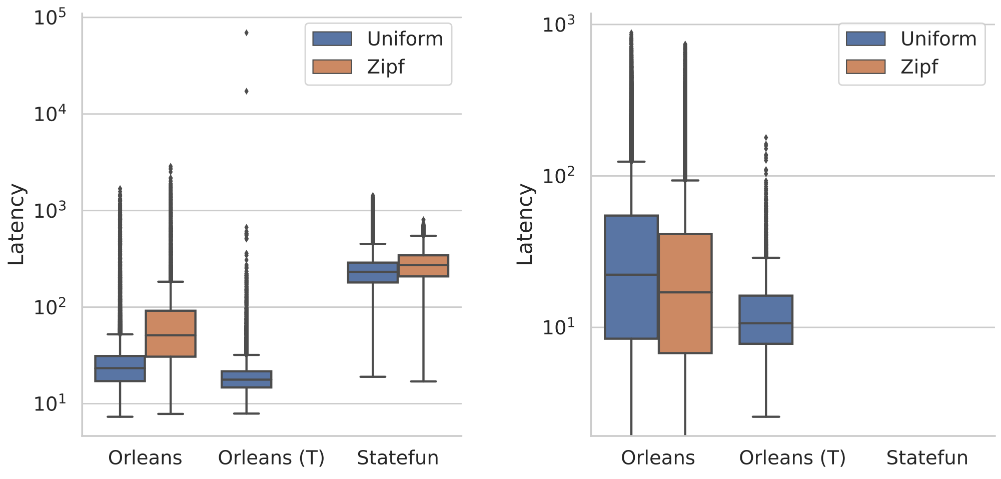
System	Distribution	Mean latency (ms)	95%	99%	Mean throughput (req/s)	Errors
Orleans	Uniform	43.7	158.0	285.6	486.7	0.0%
Orleans	Zipf	32.7	116.1	211.7	430.0	0.0%
Orleans (T)	Uniform	13.0	27.3	48.4	12.1	0.0%

but due to other operations and request latency, the simulated throughput is often slightly lower than the amount of simulated users. Regardless, we do simulate an amount of users that is 80% of the measured maximum throughput, but report the resulting measured average throughput. All stable throughput experiments are run for 10 minutes, in order to let the system acclimate to the throughput. We measure latency using logs written by the API gateway layer of the implementations. For Orleans, the latency measured is the time from when a request enters the gateway to when the gateway returns the result to the HTTP client. For StateFun, the latency measured is the time from when the request enters the gateway, to when the gateway reads the egress message linked to the request from Kafka, which indicates the request is finished.

Results question 1

> What is the mean latency of a request when the system is running the benchmark workloads under a stable load, and how does the random distribution used in the workload impact the latency?

In Table 6.1 and Figure 6.3a, latency statistics of the checkout workload at stable throughput is shown. In Table 6.1, the mean latency per checkout request is shown, together with tail latency at the 95th and 99th percentile. We see that on average, StateFun has a significantly higher latency when compared to the Orleans implementations, regardless of the distribution that is used. Orleans has a significant increase in latency when a zipfian distribution is used compared to the uniform distribution, while StateFun sees no significant difference in latency when the distribution is changed. Interestingly, the 99th percentile tail latency of Orleans with the zipfian distribution used is higher than the same tail latency for StateFun. When comparing the base Orleans implementation with the transactional variant, we can see that the transactional overhead increases the average request latency by 74.6%. In the boxplot in Figure 6.3a, we see that there are many outliers at higher latencies for all



(A) Checkout benchmark

(B) Analytics benchmark

FIGURE 6.3: Boxplot showing measured latency of requests while running the checkout and analytics benchmarks with a stable throughput. The Orleans implementation which uses ACID transactions is labelled as "Orleans (T)"

systems and distributions. However, the outliers for the Orleans applications are relatively of much higher latency than the outliers for the StateFun application.

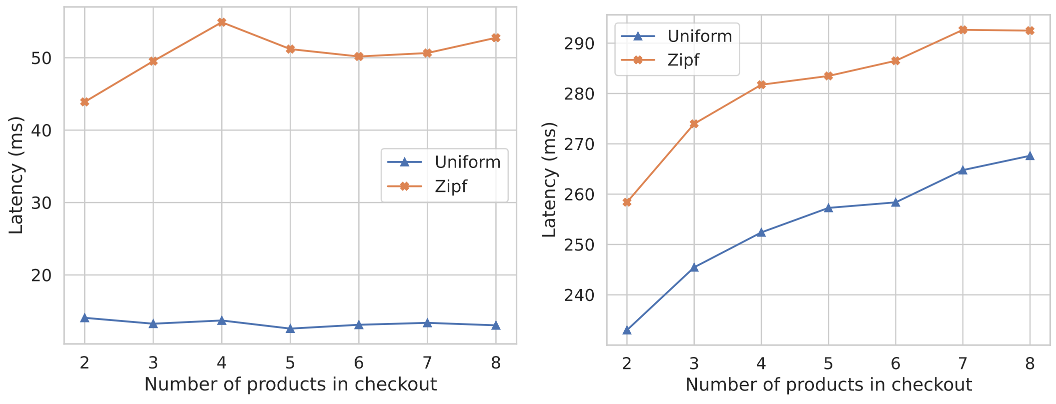
In Table 6.1, we also report the percentage of requests that do not return a successful result, under the errors column. For Orleans, this is the percentage of requests that do not return a 200 status code to the HTTP client. In StateFun, the API gateway can not wait for requests to finish on the StateFun cluster before returning, so it always returns a successful response after it forwards the request to Kafka. Therefore, we define errors for StateFun as the amount of requests that did not return an egress message while the benchmark was running. Interestingly, we see that StateFun does not respond to a large percentage of requests, 16.42% when the uniform distribution is used and 4.06% when the Zipfian distribution is used. In contrast, the error rate for Orleans is much lower, with 0.08% of requests returning a non-200 response when the uniform distribution is used.

In Table 6.2, we show latency statistics of the stable throughput experiments with the analytics workload. We see that there is a slight decrease request in latency when the zipfian distribution is used, however as discussed in subsection 6.2.1, there is not a large impact of the usage of the zipfian distribution in the analytics workload, as the analytics data is uniformly populated. In the boxplot in Figure 6.3b, we again see that there are many outliers in the requests of the Orleans and transactional Orleans experiments.

Results question 2

> How does the amount of products included in an order influence the latency of the checkout request?

In Figure 6.4, we show the mean latency of checkout requests per number of products included in the checkout. As specified earlier, checkout users add a randomly chosen amount of products between 2 and 8 in their shopping cart before requesting



(A) Orleans implementation (B) StateFun implementation

FIGURE 6.4: Mean request latency per number of products in checkout

a checkout. In the checkout workflow shown in Figure 3.1, all products are called in a fan-out pattern. Ideally, when more products are called, no difference in latency should be observed, as the product calls should be concurrent. In the Orleans implementation (Figure 6.4a), we see no discernible trend of an increased latency when more products are included in the checkout, when a uniform distribution is used. For the zipfian distribution experiment, we do see a slight increase in latency when more products are added. For StateFun, the results are shown in Figure 6.4b. For both distributions, we see a clear increase in request latency when more products are added. This shows that for StateFun, there is some overhead when functions are called concurrently.

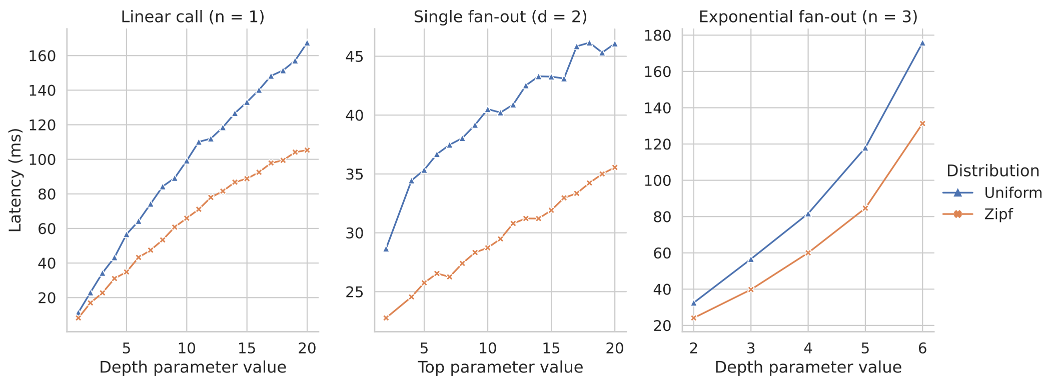


FIGURE 6.5: Mean orleans analytics request latency at stable throughput load in three request structure cases where one of the two request parameters (top or depth) is fixed. Latency is plotted for each value of the remaining parameter.

Results question 3

How do the analytics request parameters influence the request latency?

In Figure 6.5 we show the influence of the structure of the analytics request on its latency. The results are shown only for the Orleans implementation, as it was the only system that was able to run a stable throughput of the analytics workload. We highlight three cases where either the top or depth parameter is fixed and show the latency per each value of the remaining parameter. In the first case, the top parameter is set to 1 ($n = 1$), which shows the requests where products are called sequentially. We see that when the depth of the query is increased, and thus the amount of products that are called sequentially is increased, we also see a steady increase in request latency.

In the second case, the depth parameter is fixed to 2 ($d = 2$), which shows the requests where products are called in a fan-out pattern instead of sequentially. We see a slight increase in latency when the top parameter is increased, representing an increase in concurrently called products. This increase is much lower than in the linear call case, which shows that the Orleans deployment is able to run these product invocations in parallel.

The third case shows an exponential fan-out request structure, where the top parameter is fixed to 3 ($n = 3$). This is a combination of sequential and concurrent calls to products. The request latency is shown for each depth value between 2 and 6. In this case, the amount of products called in total during the request is equal to $\frac{1}{2}(3^d - 1)$ where d is the depth parameter. We observe an increase in latency as the depth parameter is also increased, which is slightly exponential. This is expected when compared with the other two cases.

6.2.3 Networking

To get a better understanding of how the tested systems work, we inspect the origin of the request latency. First, we show how much each layer in the application (API, functions and networking) contributes to the overall request latency. To gain a better understanding of the effects of request structure characteristics on latency as shown in subsection 6.2.2, we measure the contribution of networking on latency for these varying characteristics.

Experiment questions

1. How much does each application layer (API gateway, functions, and networking) contribute to request latency?
2. How does the amount of products in an order influence the contribution of networking on latency of the checkout request?
3. How do the analytics request parameters influence the contribution of networking on latency?

Experiment design

The measurements for these experiments are based on the same setup as the stable load latency experiments in subsection 6.2.2. Since we pass a tracing identifier with each internal request, we can follow the path of a request in the logs and calculate the amount of time spent in each of the application layers. If a request contains concurrent calls, the longest path is used in the calculation. For example, in the fan-out pattern in the checkout operation, the product function execution that returns a message to the order function at the latest time, is used to calculate the latency components.

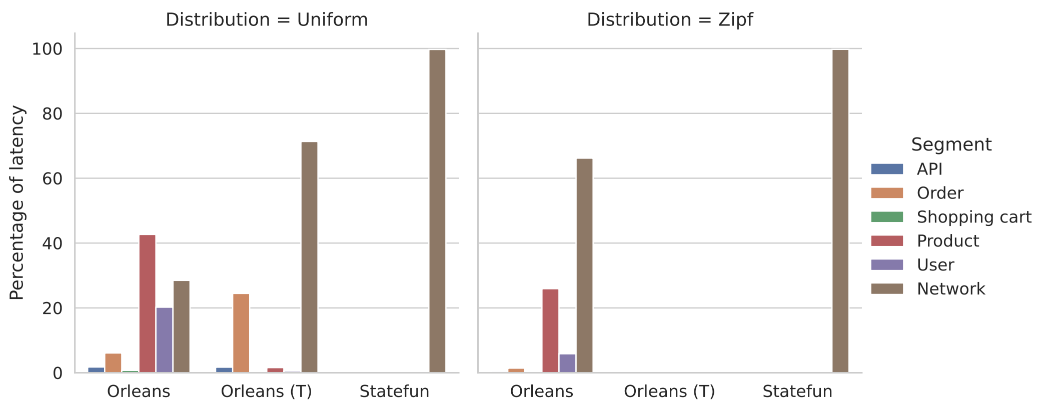


FIGURE 6.6: Mean percentage of latency in checkout request contributed by each function, the API gateway, and network, under stable throughput.

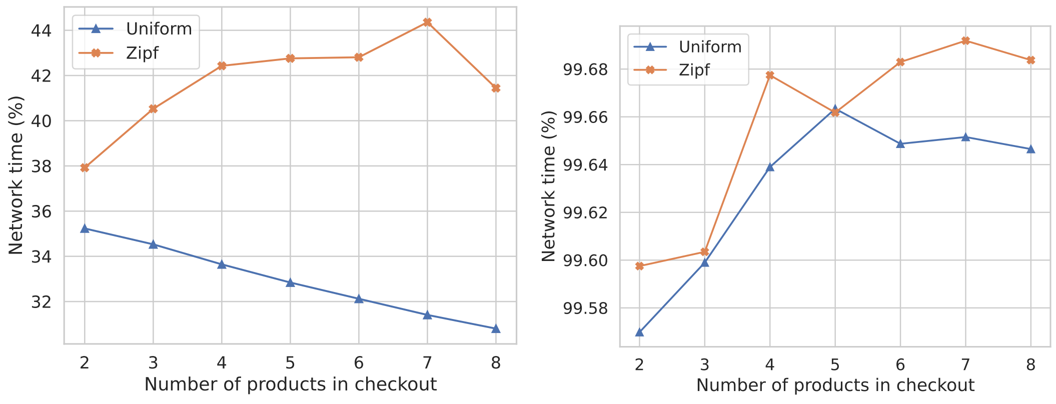
Results question 1

> How much does each application layer (API gateway, functions, and networking) contribute to request latency?

In Figure 6.6 we show for each of the implemented functions and the gateway, how much they contribute to the end-to-end latency of the checkout request, on average. All time between the measured function calls is shown as network time. These measurements are taken at stable throughput. As the checkout request contains a fan-out call to all products, we follow the longest path per request, i.e. the product call that returns the latest. We can clearly see that request latency for StateFun is almost completely classified as network time. This is expected, as StateFun does all state processing before and after the functions are called. State is shipped to the function before it is executed, and the state that is returned by the function is then stored after it stops executing. As we can only produce logs in the function code, we can not trace the latency that the state management in the StateFun runtime produces. For the Orleans implementation, we can more clearly see which data entities contribute to request latency. When the state is updated in an Orleans grain, a database write is done automatically. Therefore, the latency cost of updating database state is included in our function latency measurements. We see, that under the uniform distribution, the functions that contribute to a significant portion of the latency, are the Product and User functions. These are also the only functions that update state during the checkout process. For the transactional variant of Orleans, we see that network latency is much more prominent. This is likely due to the protocol for transactions that sends many requests to participating grains in the background. We also see that for the normal Orleans implementation, network latency is more prominent when the zipfian distribution is used in the workload. This is likely due to the fact that requests need to wait for popular product grains to become available for execution.

Results question 2

> How does the amount of products in an order influence the contribution of networking on latency of the checkout request?



(A) Orleans implementation (B) StateFun implementation

FIGURE 6.7: Mean network latency percentage for each amount of products in checkout measured under stable throughput.

In Figure 6.7 we compare the mean contribution of networking on latency to the amount of products included in the checkout request. In Figure 6.7a we see the results for the Orleans implementation. When a uniform distribution is used, we observe a slight decline in the percentage of network time. In this case, networking is not a limiting factor in concurrent calls within a request. In contrast, when the zipfian distribution is used we observe an increase in the percentage of network time. When more products are included in a checkout, the probability increases that a product is chosen that is very popular. Therefore, it is likely that this relative increase in network time is due to longer waiting times for a product grain to become available for invocation. We can only measure when the invocation of a grain starts, and so this waiting time is classified as network time.

In Figure 6.7b we show the results for the StateFun implementation. As discussed, we cannot sufficiently distinguish network time and function state management time in our logging. Therefore, almost all measured latency is due to networking. We can see a slight rise in the percentage of network and state management time when more products are processed in the checkout. This is expected, since the actual stateless function execution which is the only other contributing factor in latency, is easily parallelizable and should not increase the latency with more products.

Results question 3

> How do the analytics request parameters influence the contribution of networking on latency?

In Figure 6.8 we show how the analytics request characteristics influence the contribution of networking to request latency. Like in question 3 of subsection 6.2.2, we show three cases of varying request calling structures of the analytics benchmark run on the Orleans implementation. First, we show the linear call request structure ($n = 1$). We see that the percentage of network time increases when the depth parameter increases. When the amount of sequential calls is low, the contribution of latency from the API layer is more significant and thus the percentage of network time is lower. Since this workload is read-only, Orleans does not need to contact the database during each product invocation, which means that the product latency for each invocation is relatively low. When more products are called sequentially, the

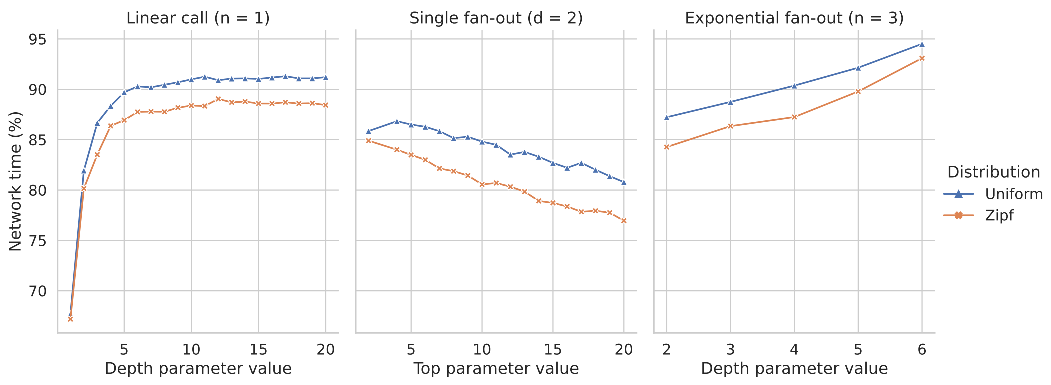


FIGURE 6.8: Mean network latency percentage of Orleans analytics requests at stable throughput in three request structure cases where one of the two request parameters (top or depth) is fixed. The mean network percentage is plotted for each value of the remaining parameter.

absolute network latency increases linearly and therefore we see an increase in the relative amount of network latency.

In the second case, we look at a single fan-out pattern ($d = 2$). We see that the percentage of network time decreases when more products are called concurrently in the single fan-out. Since there is an implicit fan-in pattern directly after the fan-out, the latency is dependent on the product entity that takes the longest to respond. When the amount of products that are called concurrently increases, the chance increases that a product is called that takes longer to respond. The absolute network latency remains relatively steady when more products are called, and therefore the relative amount of network time decreases. Furthermore, when more products are called, more data is processed in the product invocations and in the API layer, slightly increasing their latency.

The third case shows an exponential fan-out pattern where the top parameter is set to 3 ($n = 3$). We observe an increase in the share of network time when depth is increased. This means that for each depth increase, the network latency increases at a faster rate than the product latency. As the amount of products called in the request increases exponentially, this means that there is more data to transmit between the functions, which can explain the higher increase in network latency.

6.2.4 Cost

An important aspect of cloud and serverless computing is resource usage, since it directly influences the cost of running an application. Therefore, we look at the CPU usage of each of the systems when we run our benchmark application under a stable load.

Experiment questions

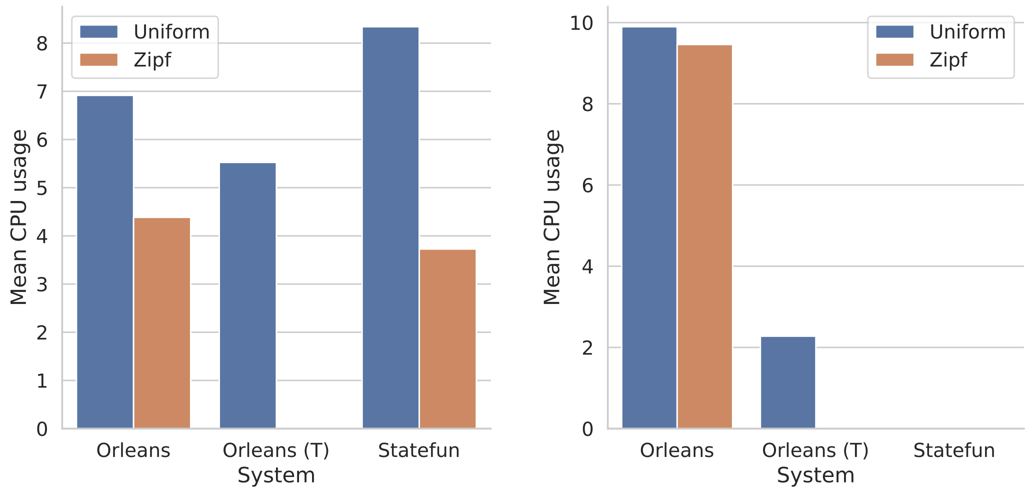
1. How much CPU resources do the systems use when running the benchmark at a stable load?
2. What is the ratio of achieved throughput to CPU usage?

Experiment design

For this experiment, the same setup as the stable load latency experiment subsection 6.2.2 is used. CPU core usage statistics are requested from Google Cloud, for the time that the experiment was running. Google Cloud runs a managed instance of Prometheus[4], which collects the CPU core usage from the kubelet processes that run on every node in the Kubernetes cluster.

Results question 1

> How much CPU resources do the systems use when running the benchmark at a stable load?



(A) Checkout benchmark (B) Analytics benchmark

FIGURE 6.9: Mean cluster CPU core usage while running stable throughput experiments.

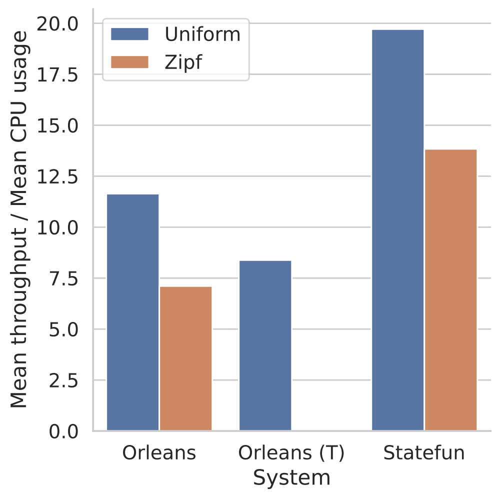
In Figure 6.9 we show the mean CPU core usage over the entire cluster of 5 `n2-standard-4` machines with 4 vCPUs each. This includes the resource usage of all background Kubernetes pods running on the system, including the log collection processes and Kafka instance which all deployments share. We see that there is a drop in CPU usage when comparing the zipfian distribution experiments with the uniform distribution experiments. This is expected since the zipfian workload is less easily parallelized over multiple machines and threads, which is also reflected in the measured throughput in Table 6.1. Furthermore, we see a drop in CPU usage when Orleans is run in transactional mode. This shows that a lack of CPU resources is not the main cause for the lower throughput.

Results question 2

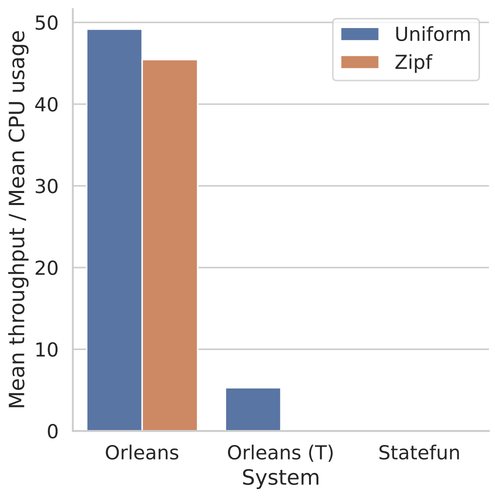
> What is the ratio of achieved throughput to CPU usage?

In Figure 6.10 we show the ratio of stable throughput achieved per vCPU core used. We can see that for the checkout benchmark, StateFun is able to achieve better stable throughput per used vCPU core. While the CPU usage drops significantly when a

[4]`https://prometheus.io`



(A) Checkout benchmark



(B) Analytics benchmark

FIGURE 6.10: Ratio of achieved stable throughput per vCPU core used.

zipfian distribution is used instead of a uniform distribution, the achieved throughput per vCPU core is still lower for both Orleans and StateFun. The transactional deployment of the Orleans application achieves a lower throughput per vCPU core than the non-transactional deployment. This is expected since the transactional protocol adds more message processing per request.

6.3 Scalability

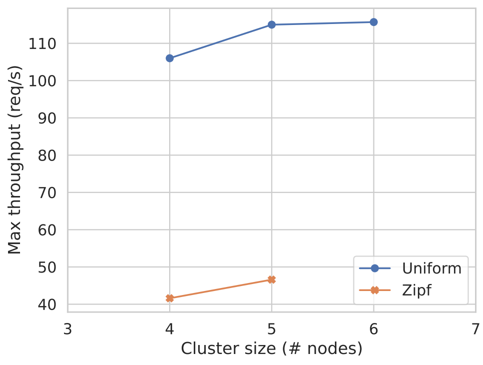
Most cloud-serving systems are designed to be horizontally scalable, in order to serve very large workloads. This is also an important property of serverless systems, as the cloud resource usage of such systems should scale with demand. We inspect the horizontal scalability of the selected systems by measuring how the maximum achieved throughput changes when more resources are added.

Experiment question

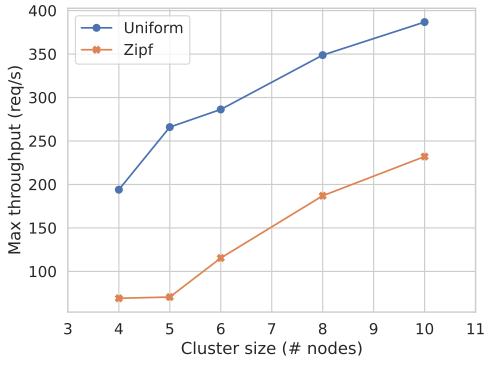
How does the amount of resources of the cluster running the benchmark application affect the maximum throughput for the checkout workload?

Experiment design

In each of our scalability experiments, we change the amount of nodes in the Kubernetes cluster. In order to determine the optimal deployment for each node configuration, we look at bottlenecks in the previous deployment. Bottlenecks are determined by the CPU usage of the pods running on the cluster, which we collect for each experiment. For example in the case of StateFun, if we observe that the Flink workers are using close to the maximum of their respective CPU resources, we deploy an extra worker instance on the node that is added in the next experiment. Our deployment of Orleans uses PostgreSQL as a state backend, which unfortunately cannot be horizontally scaled to increase write throughput without implementing sharding logic. Instead, we vertically scale the node that is running PostgreSQL. For example, instead of adding another `n2-standard-4` node, we change the `n2-standard-4` node running PostgreSQL to a `n2-standard-8` node with double the CPU and memory resources. Virtually, we consider this as if another node was added. For each cluster size that is tested, we determine the maximum throughput in the same way as in subsection 6.2.1.



(A) Orleans implementation



(B) StateFun implementation

FIGURE 6.11: Max throughput per cluster size in nodes.

TABLE 6.3: Scalability experiments with amount of `n2-standard-4` equivalent nodes, measured maximum throughput and deployment configuration.

System	Distribution	Nodes	Max throughput (req/s)	Deployment
Orleans	Uniform	4	106.0	2 orleans + 1 postgres
Orleans	Uniform	5	115.0	3 orleans + 1 postgres
Orleans	Uniform	6	115.7	4 orleans + 1 postgres
Orleans	Zipf	4	41.6	2 orleans + 1 postgres
Orleans	Zipf	5	46.6	3 orleans + 1 postgres
Statefun	Uniform	4	194.0	2 functions + 1 worker
Statefun	Uniform	5	266.0	2 functions + 2 worker
Statefun	Uniform	6	286.3	3 functions + 2 worker
Statefun	Uniform	8	348.8	4 functions + 3 worker
Statefun	Uniform	10	386.8	6 functions + 3 worker
Statefun	Zipf	4	69.1	2 functions + 1 worker
Statefun	Zipf	5	70.5	2 functions + 2 worker
Statefun	Zipf	6	115.4	3 functions + 2 worker
Statefun	Zipf	8	187.0	4 functions + 3 worker
Statefun	Zipf	10	232.1	6 functions + 3 worker

Results

In Figure 6.11 we show the measured maximum throughput per cluster size in `n2-standard-4` equivalent steps. The experiments are described in Table 6.3, which shows for each experiment how many nodes are used for each process type. In the case of Orleans, it shows how many nodes are used to run Orleans silos and what the node equivalent is for the machine that runs PostgreSQL. For StateFun, it shows how many nodes are used to run the function containers, and how many nodes are used to run StateFun Flink workers. It should be noted that each node that runs a function container also runs an API gateway container. For StateFun, we see a steady increase in maximum throughput when more nodes are added. Unfortunatly, we could not get Orleans to scale beyond the base deployment. Likely due to bugs in the PostgreSQL connector, Orleans opened too many connections to PostgreSQL, which resulted in many exceptions when running a higher amount of request throughput. It is likely that Orleans is able to scale much better when another backend for persistence is used, like Azure Storage or Amazon DynamoDB. However, we were not able to test Orleans with these configurations.

6.4 Development difficulty

While performance is an important metric to compare stateful functions systems with, the difference in the difficulty of implementing an application on a system is important as well. To compare the selected systems in this regard, we first compare the lines of code used to implement our benchmark application on each system. Lines of code (LOC) can give an indication of the amount of effort necessary to implement an application.

TABLE 6.4: Lines of code (LOC) per implementation

Implementation	LOC (Total)	LOC (Excl. gateway)	Languages
Orleans	753	536	C#
Orleans (T)	789	559	C#
StateFun	971	705	Kotlin
Cloudstate	852	663	Kotlin, Protocol Buffers
Cloudburst	480	328	Python

6.4.1 Lines of code (LOC)

We show the LOC for each of the implementations in Table 6.4. We show two values for each implementation, the LOC including and excluding the API gateway, which is implemented for each system, but does not influence the application logic. We see that the Cloudburst implementation uses the least amount of code, which is partly because Python is a very concise language when compared to C# and Kotlin. Furthermore, as described in chapter 5 the Cloudburst implementation does not share all features included in the other implementations.

The Cloudstate and StateFun implementations both have a similar amount of code, which is the most of all the implementations. They are both implemented in Kotlin, which is a JVM-based language that is generally more concise than Java. Therefore, if they would be implemented in Java, for which for example the StateFun SDK is specifically written, the LOC would be even slightly higher. The StateFun SDK requires a relatively large amount of code to implement the communication between stateful functions. For example, StateFun requires you to manually send messages to other functions and to forward incoming messages to a specific function to the right action. In comparison, in Orleans all communication is abstracted away and calls to other functions are disguised as normal procedure calls. In Cloudstate, a lot of boilerplate code is necessary to implement the event-sourced functionality of the event-sourced entities. Each operation on an event-sourced entity requires a command handler which handles the operation and sends out an entity change event if the entity should be changed in the operation. Each entity change event requires an event handler which updates the local state of the entity. Finally, an event-sourced entity requires a handler which can create a snapshot of the state, and a handler which can recreate the state of the entity from a stored snapshot. Naturally, this is an effect of the state management in Cloudstate, however, the SDK could offer an abstraction for this that would make implementing entities much easier.

TABLE 6.5: Subjective rating of programming model and system understandability

System	Programming model	System understandability
Orleans	★★★★★	★★★★☆
Orleans (T)	★★★★☆	★★★☆☆
StateFun	★★★☆☆	★★★☆☆
Cloudstate	★★★☆☆	★★☆☆☆
Cloudburst	★★☆☆☆	★★☆☆☆

6.4.2 Subjective rating

In Table 6.5 we give a subjective rating to each of the selected systems on the ease of use of the programming model, and the system understandability. The lowest rating for the programming model is given to Cloudburst, which regardless of its limited set of features was hard to understand, partly due to missing or incorrect documentation. Furthermore, the SDK doesn't allow for a clear distinction between function instances and therefore doesn't include complex communication between function instances. Therefore, its programming model doesn't fully conform to the stateful FaaS model. We also give Cloudburst a low rating for system understandability, since it is difficult to understand how important properties like function scheduling and state caching work and how they influence the application.

We have given the maximum rating to the programming model of Orleans since it is very easy to use since a large portion of the data management and communication is abstracted away. While it is only possible to implement Orleans applications in C#, it is very well integrated and easy to use for programmers already familiar with normal C# and .NET development. Furthermore, the SDK has a lot of well-documented features that do not exist on the other selected systems. For example, Orleans allows to set a request context which can be accessed by any grain included in a request, regardless of where the grain is executed. We have used this to implement distributed tracing. It's not very difficult to understand how Orleans works, especially for people familiar with distributed systems. Many properties of the system are well-documented and can be changed by the programmer, such as how grains are scheduled. Therefore, we have also given a high rating to the system understandability of Orleans.

The transactional mode of Orleans is slightly harder to understand, as it uses an elaborate protocol to run the distributed transactions. Furthermore, the transactional API of the SDK is more limited than the non-transactional API. For example, if the state of a grain is marked as transactional, every CRUD operation on that state is run as a transaction. There is no way to mark a request as non-transactional. This can be seen in our analytics request performance results, which show that the transactional Orleans is less performant than normal Orleans, while the grain reads that are performed do not have to be run as a transaction. For these reasons, we have given the transactional variant of Orleans a slightly lower score when compared to normal Orleans.

As discussed, the SDK for Cloudstate requires the programmer to write a lot of boilerplate code to implement event-sourced entities. However, the basic functionality of the SDK was reasonably documented and easier to understand than the Cloudburst SDK. Therefore we have given it a slightly higher rating than Cloudburst, but lower than Orleans.

Our rating for the StateFun programming model is the same as for Cloudstate. While the data management of function state is less work to implement when compared to Cloudstate, the communication between functions requires more code in StateFun, as discussed earlier in subsection 6.4.1. Furthermore, because of the asynchronous messaging between functions, operations can become more complex to implement. An example of this is the checkout operation in the StateFun implementation, of which the messaging between the various functions is shown in Figure 5.2. For system understandability, we rate StateFun slightly lower than Orleans, and slightly higher than Cloudstate and Cloudburst. StateFun is built on the well-documented and widely-used Apache Flink, which improves the understandability of the system. However, some internal aspects of the system, such as the origin of the relatively high request latency, are still difficult to understand due to the lack of internal tracing capabilities. Therefore, we have ranked the system understandability as slightly higher than Cloudstate and Cloudburst, but slightly lower than Orleans.

Chapter 7

Discussion

In the previous chapter, we have evaluated our benchmark design by conducting various experiments. Firstly, we have shown that it can be used to evaluate the general performance of a stateful functions system, and to compare throughput and latency between different systems. Furthermore, we have shown that the benchmark workloads can be used to analyse various properties of stateful functions systems, such as the impact of networking on latency and the resource usage of these systems while under a reasonable load. Finally, we have shown that the dynamic request structures of our benchmark workloads allow us to analyse how the systems cope with concurrency in different calling patterns, such as the fan-out and exponential fan-out.

7.1 Limitations

Unfortunately, our analytics benchmark was only able to run on Orleans, and not on the StateFun system. This was probably caused by the relatively high number of internal requests per operation, especially in the exponential fan-out cases. The workload could be tweaked to reduce the number of internal requests. However, we observed that Orleans was able to achieve a significant throughput with the same workload, and thus we consider the workload to be a valid stress test for stateful functions systems. Therefore, we consider the fact that the workload could not be run on the StateFun system as a flaw of StateFun, and not as a flaw of the benchmark.

Our analysis of network latency in subsection 6.2.3 was limited in the case of StateFun, since we were not able to properly analyse the internal workings of the StateFun runtime. This restricted us to classify the time a request spent outside of our implementation as network latency, while the data management by the StateFun was done in this time as well. An important recommendation to stateful functions systems is to include the capability to trace a request throughout the system and the application, which would allow for better profiling of request latency. Orleans includes the option to store values in a request context, which is then available in any grain participating in the request. This allowed us to set and log an identifier for each request, allowing us to trace the request over the system. For StateFun we could only manually add a request identifier to each internal message to be able to perform limited request tracing. Better support for this in stateful functions frameworks would help to further understand these systems.

A general limitation of our benchmark is that benchmarking a new system requires a specific implementation of the benchmark application for that system. In contrast to benchmarks such as YCSB (Cooper et al., 2010) and TPC-C (Transaction Processing Performance Council (TPC), 2010), where only a thin driver layer between the benchmark client and database is required, this requires significantly more time. However, due to the nature of stateful functions, where state management and application logic is combined, we see no alternative to this approach.

7.2 Future work

Our benchmark is mostly focused on measuring the performance of stateful serverless functions. Further research into benchmarking stateful functions could explore evaluating other aspects of these systems, such as transaction consistency. When more stateful functions systems provide ACID transactions, as for example Orleans does, benchmarks should test for inconsistencies. YCSB-T (Dey et al., 2014) does this by including a validation check after the benchmark has run to measure inconsistencies and give an anomaly score.

Another property of stateful functions systems that could be evaluated by future benchmarks is their ability to scale while running a workload. YCSB includes a metric called *elastic scaleup*, which measures the performance improvement of a database system when an extra server is added while a workload is running. Ideally, a stateful functions system should be able to dynamically reconfigure while running, and show a performance improvement quickly after a server is added. A closely related property that can be evaluated is autoscaling, where the system automatically provisions new instances according to demand.

Chapter 8

Conclusion

In this thesis, we have introduced a benchmark specifically designed for Stateful Functions-as-a-Service (SFaaS) systems. The benchmark exists of a benchmark application based on a simple e-commerce application that handles products, users, shopping carts and orders, and a benchmark client which runs two benchmark workloads. The benchmark workloads call complex application operations that span multiple stateful functions and are designed to test many different function calling patterns. In this way, it differs from existing cloud systems benchmarks which are not focused on SFaaS systems. We have implemented the benchmark application on four selected stateful functions systems: Microsoft Orleans, Apache Stateful Functions, Cloudstate and Cloudburst. We have evaluated our benchmark by comparing these selected systems, with a range of experiments on cloud deployments of our application implementations. We show that our benchmark can be used to compare the performance of SFaaS systems, in the form of request latency and throughput. Furthermore, we show that with the varied function calling structures of our workload, we can observe how request latency changes with concurrent and subsequent function calls, and what the influence of internal networking is on this latency. By observing cloud resource usage and request throughput, we show that our benchmark can also highlight the cost of running applications that are deployed on stateful functions systems. Finally, we show that the scalability of the systems can be evaluated by observing the maximum request throughput when more computing resources are added to the deployments. While we have designed our benchmark to test the performance of stateful functions systems, we have also designed it to be extendable by future work to test a wider range of properties.

Appendix A

Application API Schema

A.1 Product

A.1.1 Create new product

Route: **POST** `\products`

Body (JSON):

Name	Type
price	Integer (optional)
stock	Integer (optional)

Returns String

Description Adds a new product to the system and returns the ID of the new product.

A.1.2 Update product

Route: **POST** `\products\{id}`

Body (JSON):

Name	Type
price	Integer (optional)
stock	Integer (optional)

Returns String

Description Updates the product with the given id.

A.1.3 Query frequently bought together items

Route: **GET** `\products\{id}\freq-items`

Query params:

Name	Type
top	Integer (optional)
depth	Integer (optional)

Returns List of strings

Description Query frequently bought together items of the product with the given id. Uses the given query parameters as described in subsection 3.2.2.

A.2 User

A.2.1 Create new user

Route: **POST** `\users`

Body (JSON):

Name	Type
credits	Integer (optional)

Returns String

Description Adds a new user to the system and returns the ID of the new user.

A.2.2 Add credits to user

Route: **PATCH** `\users\{id}\credits\add`

Body (JSON):

Name	Type
credits	Integer

Returns String

Description Adds credits to the users with the given id.

A.3 Shopping cart

A.3.1 Create new shopping cart

Route: **POST** `\shopping-carts`

Returns String

Description Creates a new shopping and returns the ID of the new shopping cart.

A.3.2 Add product to shopping cart

Route: **POST** `\shopping-cart\{id}\products`

Body (JSON):

Name	Type
productId	String
amount	Integer

Description Adds a given amount of the product with the given productId to the shopping cart.

A.4 Order

A.4.1 Checkout order

Route: **POST** `\orders\checkout`

Body (JSON):

Name	Type
cartId	String
userId	String

Description Run checkout for an order with the given shopping cart and user, as described in subsection 3.2.2.

Appendix B

Source code

The source code of the benchmark application implementations for Orleans, Apache Stateful Functions, Cloudstate, and Cloudburst, and the implementation of the benchmark client, can be found on GitHub via the following link: `https://github.com/mcomans/stateful-functions-benchmark`.

Bibliography

Armbrust, Michael et al. (Feb. 2009). "Above the Clouds: A Berkeley View of Cloud Computing". In: p. 25.

Baldini, Ioana et al. (2017). "Serverless Computing: Current Trends and Open Problems". In: *Research Advances in Cloud Computing*. Ed. by Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya. Springer, pp. 1–20. DOI: 10.1007/978-981-10-5026-8_1.

Bernstein, Phil et al. (Mar. 2014). *Orleans: Distributed Virtual Actors for Programmability and Scalability*. Tech. rep. MSR-TR-2014-41.

Carbone, Paris et al. (2017). "State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing". In: *Proc. VLDB Endow.* 10.12, pp. 1718–1729. DOI: 10.14778/3137765.3137777.

Cooper, Brian F et al. (2010). "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*, pp. 143–154.

Copik, Marcin et al. (2021). "SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing". In: *Service Computing*, p. 15.

Dey, Akon et al. (Mar. 2014). "YCSB+T: Benchmarking Web-Scale Transactional Databases". In: *2014 IEEE 30th International Conference on Data Engineering Workshops*. Chicago, IL, USA: IEEE, pp. 223–230. ISBN: 978-1-4799-3481-2. DOI: 10.1109/ICDEW.2014.6818330.

Eldeeb, Tamer and Phil Bernstein (Oct. 2016). *Transactions for Distributed Actors in the Cloud*. Tech. rep. MSR-TR-2016-1001.

Gan, Yu et al. (2019). "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud &amp; Edge Systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, pp. 3–18. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304013.

Hellerstein, Joseph M. et al. (2019). "Serverless Computing: One Step Forward, Two Steps Back". In: *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org.

Jonas, Eric et al. (2019). "Cloud Programming Simplified: A Berkeley View on Serverless Computing". In: *CoRR* abs/1902.03383. arXiv: 1902.03383.

Raab, Francois Raab, Walt Kohler, and Amitabh Shah (1992). *Overview of the TPC-C Benchmark*. Tech. rep.

Sreekanti, Vikram et al. (Aug. 2020). "Cloudburst: Stateful Functions-as-a-Service". In: *Proceedings of the VLDB Endowment* 13.12, pp. 2438–2452. ISSN: 2150-8097. DOI: 10.14778/3407790.3407836.

Transaction Processing Performance Council (TPC) (Feb. 2010). *TPC Benchmark C, Standard Specification, Revision 5.11*. Tech. rep.

– (Apr. 2015). *TPC Benchmark E, Standard Specification, Version 1.14.0*. Tech. rep.

Wang, Liang et al. (2018). "Peeking behind the Curtains of Serverless Platforms". In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA,* *July 11-13, 2018*. Ed. by Haryadi S. Gunawi and Benjamin Reed. USENIX Association, pp. 133–146.

Wu, Chenggang, Vikram Sreekanti, and Joseph M. Hellerstein (Feb. 2019). "Autoscaling Tiered Cloud Storage in Anna". In: *Proceedings of the VLDB Endowment* 12.6, pp. 624–638. ISSN: 2150-8097. DOI: 10.14778/3311880.3311881.

Wu, Chenggang et al. (2019). "Anna: A KVS For Any Scale". In: *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1. ISSN: 1041-4347, 1558-2191, 2326-3865. DOI: 10.1109/TKDE.2019.2898401.