KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Distributed Systems

Second cycle, 30 credits

The State of Stateful Serverless Systems

ASHMITHA AMBASTHA

Stockholm, Sweden, 2023

The State of Stateful Serverless Systems

ASHMITHA AMBASTHA

Date: January 31, 2023

Supervisors: Assistant Prof. Paris Carbone, Jonas Spenger
Examiner: Prof. Vladimir Vlassov
School of Electrical Engineering and Computer Science
Host company: RISE AB
Swedish title: Tillståndet av Stateful Serverless System

Abstract

Serverless computing and Function-as-a-service are popular paradigms that simplify application development by abstracting the development layer from the underlying infrastructure components. These systems work in a pay-as-you-go model and provide an efficient environment for developers to focus entirely on developing their business logic. Different stateful serverless systems are available—for example, Apache Flink StateFun, Azure Durable Functions, Kalix, and Cloudburst. Currently, there is a lack of comparison criteria to compare these systems successfully.

In this thesis, we approach solving the question of how we can compare different stateful serverless systems. We devise a comparison criterion and assess four state-of-the-art systems. From these comparison results, we derive assessments that ease choosing one system over the other in the best interest of the user's development goals. These results and assessments can be applied to more upcoming stateful serverless systems and attempt guiding product owners, software engineers, and researchers to choose a stateful serverless system that best suits their application development goals.

We devise a comparison criteria based on the system's architectural, functional, and performance qualities. The results of these comparisons list essential features. Based on architectural qualities, we conclude that features such as different function types and patterns are essential criteria. Based on functional qualities, we conclude that features such as being language agnostic, support for different in-built primitives, and state and message guarantees are important comparisons.

Performance quality results include locally running the chosen system and testing with varying numbers of sequential invocations. The chosen systems are Apache Flink StateFun and Azure Durable Functions. We test the system with varying numbers of sequential invocations to a Greeter function and find the system's execution time in seconds and throughput in events per second. From the results, we infer that the performance of Apache Flink StateFun is better than Azure Durable Functions when running the systems locally and invoking the function sequentially.

Keywords

Stateful Serverless Systems, Azure Durable Functions, Apache Flink StateFun, Kalix, Cloudburst, Function-as-a-Service

Sammanfattning

Serverless computing och Function-as-a-service är populära paradigmer som förenklar applikationsutvecklingen genom att abstrahera utvecklingslagret från de underliggande infrastrukturkomponenterna. Dessa system fungerar enligt en pay-as-you-go-modell och erbjuder en effektiv miljö för utvecklare som helt och hållet kan fokusera på att utveckla sin affärslogik. Det finns olika ”stateless serverless” system - till exempel Apache Flink StateFun, Azure Durable Functions, Kalix och Cloudburst. För närvarande saknas det jämförelsekriterier för att kunna jämföra dessa system på ett framgångsrikt sätt.
I den här avhandlingen försöker vi lösa frågan om hur vi kan jämföra olika stateful serverless-system. Vi utformar ett jämförelsekriterium och utvärderar fyra toppmoderna system. Utifrån dessa jämförelser drar vi slutsatser av bedömningar som underlättar valet av ett system framför ett annat i bästa intresse för användarens utvecklingsmål. Dessa resultat och bedömningar kan tillämpas på fler kommande stateful serverless-system och försöka vägleda produktägare, programvaruingenjörer och forskare att välja ett stateful serverless-system som bäst passar deras mål för applikationsutveckling. Vi utarbetar ett jämförelsekriterium baserat på systemets arkitektoniska, funktionella och prestandakvaliteter. Resultaten av dessa jämförelser innehåller en förteckning över väsentliga funktioner. Utifrån arkitektoniska kvaliteter drar vi slutsatsen att funktioner som olika funktionstyper och mönster är väsentliga kriterier. På grundval av funktionella egenskaper drar vi slutsatsen att egenskaper som att vara språkoberoende, stöd för olika inbyggda primitiver samt tillstånds- och meddelandegarantier är viktiga jämförelser.
Resultaten av prestandakvaliteten omfattar lokal körning av det valda systemet och testning med varierande antal sekventiella anrop. De valda systemen är Apache Flink StateFun och Azure Durable Functions. Vi testar systemet med varierande antal sekventiella anrop till en Greeter-funktion och finner systemets exekveringstid i sekunder och genomströmning i händelser per sekund. Av resultaten drar vi slutsatsen att prestandan hos Flink StateFun är bättre än Durable Functions när systemen körs lokalt och funktionen triggas sekventiellt.

Nyckelord

Serverlösa system utan tillstånd, Azure Durable Functions, Apache Flink StateFun, Kalix, Cloudburst, funktion som tjänst

Acknowledgments

This thesis is a culmination of months of hard work. Work of this magnitude would not have been possible without the guidance of my academic supervisor *Assistant Prof. Paris Carbone* and my RISE supervisor *Jonas Spenger*. Their constant guidance and advice has helped me immensely. Thank you for all your support and patience.

I would also like to thank my professor and thesis examiner *Prof. Vladimir Vlassov* for his fast responses whenever I needed a review and if any documentation needed his approval. I would also like to thank Vlad Sir for his trust in selecting me as a Teaching Assistant and for always being available to guide and mentor me.

I'm forever grateful to have the unwavering support of my parents and brother, who have been my pillars of strength. Thank you for constantly pushing me to strive for the best and helping me plan my career. *Ma*, *Papa*, and *Ayush* - Thank you for being my biggest cheerleaders and strongest critics.

The last two years has been a fantastic learning experience, and I have my friends and colleagues at KTH to thank. Everyone I've had the pleasure of meeting has broadened my perspective and made me a better person. I have also been fortunate to have family and friends back home in India who have been with me through thick and thin. I'm blessed to be sharing my journey with all of you.

I want to dedicate my Master's degree to my late maternal grandfather, Mr. Ramji Prasad and my late paternal grandfather Commander BB Ambastha (Retd.), for always believing in me and for their constant love, guidance, and blessings. You're both very dearly missed, *Nana ji* and *Dada ji*.

Stockholm, January 2023
Ashmitha Ambastha

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Change is the only constant, and this quote remains valid even in the world of infrastructure engineering. The world of infrastructure engineering has seen several developments over the last few years. From working with physical machines and servers to working with Virtual Machine (VM), containerizing applications, and serverless systems.

Serverless computing is a cloud computing model that makes the cloud architects and engineers responsible for managing the underlying server-level requirements, such as infrastructure and complexity. It allows the user, the application developer, to focus only on developing the business logic.

A limitation of the early adoption of serverless systems was that support was only available for stateless application development. Hence, application development teams maintained a preserving 'state' for their application using external storage solutions. Due to this limitation, development teams introduce state management to achieve a stateful application development. Maintaining and managing an external state storage component is a cumbersome, complex task and digresses from the aim of building successful, robust applications. As a solution, stateful serverless systems are introduced, for example, Apache Flink Stateful and Azure Durable Functions.

Whether stateful or stateless, these serverless systems assist in easily incorporating cloud computing into mainstream software development without introducing any hindrance to the application developers and end-users.

There is a lack of a comparison criteria aimed at comparing Stateful Serverless Systems [1]. To successfully compare and assess stateful serverless systems, it is essential to acknowledge their strengths and weaknesses. It also opens doors to possibilities of deploying these stateful serverless systems for applications that make the most of that particular system.

1.1 Background

This thesis aims to compare state-of-the-art Stateful Serverless Systems based on their architectural, functional, and performance-based features. The result of the thesis is a list of criteria and features that help choose and compare various stateful serverless systems.

1.1.1 Introduction to Serverless Systems

A serverless system or serverless computing [2, 3] is a cloud-native execution model that allows its users, i.e., generally application developers, to develop applications and then deploy them to a cloud infrastructure without having to manage the underlying infrastructure and its operational aspects. Underlying infrastructure such as servers, databases, and storage systems. The underlying systems are maintained and developed by an infrastructure management expert.

Function-as-a-Service (FaaS) explains a serverless function that can be run in containers. These containers are maintained by the infrastructure provider chosen by the development team [4].

Building an application on serverless systems enables an on-demand experience with respect to scalability, as they work on a pay-as-you-go model. The consumers (developers) get billed only when their code actually uses the serverless service and resources in its execution. Developers write code as a set of functions which is then internally forwarded to the cloud providers to execute on their servers. The application developers build code in the form of functions; hence, serverless is also referred to as FaaS. Some popular serverless systems are AWS Lambda [5], Microsoft Azure Functions [6, 7], and Apache Flink [8].

1.1.2 Introduction to Stateful Serverless Systems

Serverless systems such as AWS Lambda and Azure Functions are stateless, i.e., no state management is available for any application. A stateless application means that each run of the application has zero knowledge of any previous application run. These applications and their results only depend on the input provided by the user.

Serverless systems that support state management of the application are called **Stateful Serverless Systems**.

Stateful Serverless Systems are faster as it removes the dependencies of reading and writing into databases when working with stateful applications. The functions and data necessary to make decisions are located in the same location as the function's logic. This makes it faster as the functions do not constantly have to read and write to work in real time. [9]

Examples of stateful serverless systems are Apache Flink Stateful and Azure Durable Function. Both are built on top of the existing stateless serverless systems as an extension to support stateful application development with additional features such as defining workflows and introducing different function types and patterns etc.

This thesis will explore a variety of stateful serverless systems to develop a method of best assessing and choosing the right stateful serverless system. The aim is to devise a method to compare the systems successfully.

Building stateful applications on serverless systems have several benefits, such as Apache Flink Stateful [10] and Azure Durable Function [7]. Some note-worthy benefits are,

- **Users do not perform server management** - Application developers, users of stateful serverless systems, do not have to worry about the underlying servers where the application runs.
- **Defining workflows** - In stateful serverless systems such as Azure Durable functions, it is possible to define a workflow [6].
- **Scalability**, as stateful serverless systems work on the 'Pay-as-you-go' concept, it makes it easy to implement application scalability and leaves the onus of the application in the developer's hands. Serverless architectures are inherently scalable. [11]

1.2 Problem

With a boom in serverless systems and their development, we have a variety of systems that aim to deliver different features to support different application development and use cases. These serverless systems aim to provide a cloud-native development model to its users to build and run applications without managing the underlying servers amongst other infrastructure components. As each stateful serverless system has a different architecture with different functionalities and performance, one stateful serverless system might be better suited to support a project than another.

As there is an absence of a method specifically aimed at comparing stateful serverless systems to make the most suitable choice while choosing one system over the other, it is most reasonable to depend upon a set of comparisons that give a wholesome overview of each stateful serverless system.

1.2.1 Problem statement

The aim of this thesis is to create a method to successfully compare and assess stateful serverless systems based on their architectural, functional, and performance qualities.

1.3 Goal

This thesis aims to explore and introduce a comparison criteria to compare and assess stateful serverless systems that can be reused for other existing and new stateful serverless systems. To accurately address the goal of the thesis project, the goal has been divided into the following three sub-goals,

- Sub-goal I: **Architectural qualities** (Review Question-1)
- Sub-goal II: **Functionality qualities** (Review Question-2)
- Sub-goal III: **Performance qualities** (Review Question-3)

The three sub-goals results help users assess which stateful serverless systems best suit their application development objectives and practices.

1.4 Contributions

This thesis aims to understand the state of different stateful serverless systems by comparing their qualities. This thesis aims to contribute and assist in

- There is a lack of a method that aims at comparing stateful serverless systems [1]. This thesis contributes to building a method that can make choosing one system over the other efficient.
- The aforementioned list is built based on comparing state-of-the-art stateful serverless systems and documented for architectural and functional qualities. And by running test cases that help examine system performance.

- These results can work as a guide for Product Owners, Infrastructure Engineers, and researchers by assisting them in comparing and assessing stateful serverless systems.

With this thesis report, the contributions can be mapped with each section. The following are contributions aimed by each section in this report,

- In the Background section, the goal is to contribute by summarizing all related work and understanding the need to develop a comparison list based on the review questions.
- In the Methods section, the goal is to contribute by devising a method of comparing different stateful serverless systems and choosing appropriate features to compare and test. Also, finalize test cases that can best test different systems' performance.
- In the Results section, the goal is to contribute by implementing the methodology and achieving results to derive conclusions.
- In the conclusions section, the goal is to deduce appropriate conclusions which can create a pathway to building enhanced stateful serverless systems.

1.5 Benefits, Ethics and Sustainability

With an increase in the development of various Stateful Serverless Systems (SSS), choosing one offering over the other needs comparison criteria that can help researchers, product owners, and developers. This thesis is a stepping stone in the direction of creating comparison criteria and standards that can be followed for all new SSS. This thesis has no ethical concerns as most of the data and comparisons used to achieve results are created randomly.

The benefit of this thesis is that the most appropriate SSS can be chosen for the development of a stateful application by the application development team. By choosing the most appropriate system, computational resources can be efficiently used.

Throughout the project, there has been no utilization of personal data, and the tools used are free of any security risks to individuals or organizations. Proper referencing is mentioned throughout, and any previous work that helps this project draw conclusions and build an understanding of the concerned stateful serverless systems is clearly referenced.

1.6 Delimitations

The first limitation is based on the stateful serverless systems chosen for comparison to answer the review questions. The aim is to choose diverse yet popularly implemented systems. This helps cover a variety of architectural and design capabilities.
Secondly, as several systems are constantly under development while conducting these comparisons, this thesis aims to use the latest available release of the systems and the latest available official documentation.
Thirdly, the test cases are run locally and not in the cloud, unlike some related work on this topic. Running these systems locally introduces some resource constraints, but it gives an opportunity to run small-scale projects and highlight how they would perform on these systems.
Lastly, for performance-based comparisons, it is essential to note that the systems use very different back-end storage types. These different state storage solutions affect the performance of the system. The thesis is built on using the default state storage that the system provides. This helps create an equal playing field for each system while answering review questions.

1.7 Structure of the thesis

Chapter 2 builds an understanding of stateful serverless technologies and the need to compare these systems. This chapter helps us assess and appreciate the results gathered by this thesis.
Chapter 3 explains the methodology used to compare these systems based on the review questions. Why and how the method works and finds common grounds to equally performance test and compare the systems.
Chapter 4 comprises how the methodology explained earlier has been implemented and how each review question is answered.
Chapter 5 contains a cohesive list of comparison results and analysis derived.
Chapter 6 covers discussions on the presented results.
Chapter 7 concludes the thesis while mentioning what can be achieved in the future for comparisons between stateful serverless systems.

Chapter 2

Background

This chapter sets the background to stateful serverless systems and the need for comparison criteria for different state-of-the-art systems. Highlights the workings of each selected system and attributes that are different. Lastly, similar systems and related work are discussed.

2.1 Stateful Serverless Systems (SSS)

2.1.1 Stateful applications development

Stateful applications are capable of maintaining data about past requests and actions. This information makes the application run easier as the result of one output can be used as an input to another and can be successfully tracked when debugging the system. Hence, the overall user experience is improved.

The following are just some common scenarios for implementing stateful applications:

- For e-commerce websites, stateful applications can be used to monitor a user's shopping basket and preserve its state over several sessions. This also leads to tracking a user's preferences, which can be used to personalize a user's shopping experience.
- In the world of gaming, stateful applications can be utilized to keep records of a user's accomplishments and progress during several game sessions.
- On social media platforms, stateful applications can be used to monitor and keep a record of a user's relationships and interactions on the social media platform.

As there is a need to build stateful applications, the process is being made much easier and more efficient using stateful serverless systems.

2.1.2 Defining stateful serverless systems

Stateful serverless systems can be defined as systems that leverage serverless computing technologies and can preserve state or track data and changes to that data over time. This differs from conventional stateless systems, which consider each request a separate event and does not keep track of past interactions or requests.

2.1.3 Issues when working with stateful serverless systems

Even though stateful serverless systems offer numerous advantSages, developing and maintaining these kinds of systems can present several disadvantages and challenges. Some issues with stateful serverless systems are,

- State management across several requests and functions necessitates additional infrastructure and logic. This can make stateful systems more difficult to build and keep than stateless ones, as it can be costly. The goal of serverless systems is to minimize resource utilization of the function while it's running and when it's idle in development. [12]
- Most importantly, to retain state across several requests and processes, stateful systems normally need some data storage to save the application state. However, selecting the best data storage solution can be tricky as it needs consideration of aspects such as cost, performance, and scalability. Each storage solution brings its own benefits, drawbacks, and performance bottlenecks.
- Although serverless systems are made to be incredibly scalable and dependable, they are nevertheless susceptible to faults or breakdowns that might compromise the long-term viability of stateful systems. If you need to keep essential data or state over extended periods of time, this can be very difficult.
- **Security**: Maintaining a state may also raise new security issues since users must consider safeguarding important information and ensure strong isolation of functions is implemented since these are running on

a shared platform. [12] This leads to creating access roles within the application development framework, and parties from the development team would need to request access, etc.

Building and maintaining stateful serverless systems can be demanding, but the advantages exceed the drawbacks in many circumstances. If a development team is trying to decide if a stateful serverless system is the best option for their application, it's crucial to pay close attention if the application has unique demands and specifications. Hence, a guide on successfully comparing stateful serverless systems needs to be devised. **This thesis is an attempt to build this comparison criterion.**

2.2 Chosen SSS

There has been an increase in serverless systems and functions development and adoption. Hence, research to develop and extend features has also taken place [9, 13, 14, 15, 16, 17, 18]. For this thesis, the chosen systems are described in the following sub-sections.

2.2.1 Apache Flink StateFun

Apache Flink Stateful Functions [10], StateFun, is an Application Programming Interface (API) built on top of Apache Flink, which simplifies the building of distributed stateful applications with a run-time built for serverless architectures.

For developers and researchers, StateFun enables the development of stateful applications with a strong message and state consistency guarantees, along with Apache Flink's advantageous features, such as fault-tolerance and scalability.

StateFun runtime is built on top of Apache Flink and uses several of Flink's features and functions. For consistent stateful streaming applications, like in Flink, StateFun incorporates the technique of co-location of state and messaging in the cluster. [19]

Parts of an application include - messages to the ingresses, messages between functions, and, eventually, messages to egresses. For a StateFun application, the mentioned parts of an application are routed through a StateFun cluster, and the state is also maintained in the StateFun cluster itself, refer to Figure 2.1. Like in Flink, StateFun also has the StateFun cluster and the function state co-partitioned; hence, it has local state access.

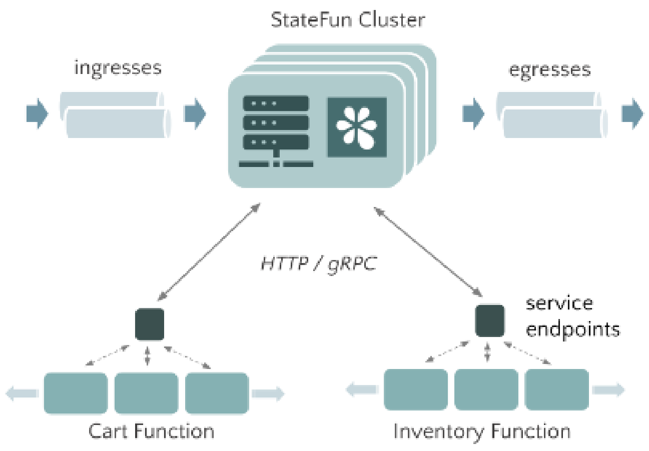


Figure 2.1: Overview of a StateFun cluster, figure reproduced from [19]

The functions are outside the StateFun cluster, but messages and invocations are routed through the StateFun cluster. Messages are sent to the functions using the function's logical addresses, which have the function type and instance ID. These logical addresses are used as the partitioning key for the message and state, this makes sure that the message reaches the correct cluster partition and hence, is locally available.

In StateFun, the computation does not happen in the cluster partitions but happens remotely in the function services using the Remote Invocation Request-Reply Protocol. Using this protocol, the StateFun cluster partition communicates with the functions in the following steps, [19]

- Step 1: The cluster partition receives an input message (ingress) and invokes the target function.
- Step 2: The service request carries the input event and checks for the current state of the function from the local state.
- Step 3: If an outgoing message (egress) and changes to the local state are being made, it is all routed through the StateFun cluster again.
- Step 4: Once the StateFun cluster receives the response, any changes to the local state are written, and the outgoing message is appropriately routed to the next cluster partition.
- The above process is done again and again until the invocation or function call ends.

When the partition is processing an event, the run-time will fetch the current state of the instance and mark it as "busy" this informs any other messages on the way for this logical address that processing is taking place. Furthermore, once completed, the logical address is marked as "available" for the following invocation, and the cycle continues. The whole state of the system and its functions are saved using Flink's snapshot mechanism. These snapshots can then be stored and rolled back in case of failures.

2.2.2 Azure Durable Functions

Azure Durable Functions (DF) [20] [7] is an extension on top of Azure Functions. They have additional features that help us perform activities such as defining the workflow of the code to implement error handling, parallel executions, and creating long-running tasks.
A function in this system is "triggered" by a specific type of azure function, for example, HTTP function, Timer function, etc. There are four durable functions types:

- Orchestrator Function - This function describes how the actions are supposed to be executed and in which order the functions are supposed to be triggered. It follows and checks with the History Table (accessed through the Azure Portal) to ensure that the triggered function is completed before another function is triggered.
- Activity Function - These are the functions that perform the actual work. If an Orchestrator function is built to process a "larger task," it can be broken into smaller tasks. Each of these tasks is an individual activity function in the world of Durable Functions.
- Entity Function - These functions define the workflows and processes for updating small pieces of state in a Durable Function execution.
- Client Function - In Durable Functions, Orchestrator functions are not triggered directly from Azure Portal. Instead, they must be triggered as part of running a Client function. These functions start the Orchestrator function and/or the entity function as part of its implementation model.

The following steps explain how a Durable Functions (DF) works,

- Step 1: The trigger process (HTTP trigger in this thesis) starts another function called the Orchestrator function. The Orchestrator function then schedules the activity function, which does the work, and the Orchestrator function is completed.

- Step 2: When the Activity function completes that task, the Orchestrator is triggered again by the Durable Task Framework to run from the beginning.
- Step 3: The Orchestrator function would again check the execution history to check whether the activity function is completed. And if yes, the response is sent, and the Orchestrator function then schedules the start of the next Activity function. This process is continued until the execution history is completed and no more Activity functions are scheduled to be triggered and run.
- Step 4: When all the function calls are complete, the Orchestrator function is started for the last time starts to check if the cleanup was successful and returns the result.

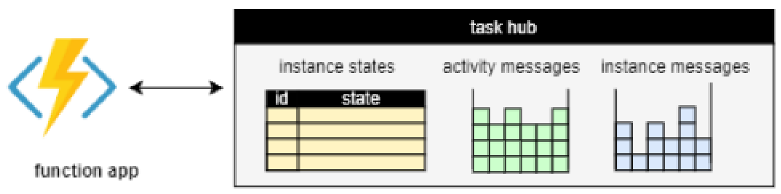


Figure 2.2: Durable Function task hub, figure reproduced from [21]

A TestHubNameHistory table associated with an Azure Storage Account represents the current state of the application in storage. It stores details on the progress of Orchestrator, Entity, and Activity functions, along with timestamps and IDs. Figure 2.2 represents the TestHubNameHistory table and its supporting tables.[21]

2.2.2.1 Durable Functions and different storage options such as Netherite

In the research paper, "Serverless Workflows with Durable Functions and Netherite" by Sebastian Burckhardt et al.[6], Netherite is a distributed execution engine built to execute serverless workflows with Durable functions efficiently.

In a default Durable Functions application implementation, the default storage is Azure Storage queues and instances in Azure Storage tables. In this case, performance metrics such as throughput are limited by how many operations the Azure storage allows per second. To improve performance in production and large-scale clusters, Netherite, an efficient execution engine, is introduced. Using Netherite is not the default configuration while working with Durable functions, so this is not part of this thesis's test cases and performance-based qualities section.

With Netherite, Durable functions achieve stateful serverless functions with high-performance metrics. The complexity of developing stateful functions is removed while building fault-tolerant, scalable applications. Architecturally Netherite represents queues and partition states in a different fashion. It has three main components,

- Persistent queues - For function communications
- Persistent storage - To maintain a state of the functions as event logs
- Compute nodes - Functions are executed on compute nodes

Reliable tracking of message queues and overall states creates many instances in default DF, which in turn creates a significant overhead on the system. Netherite's approach to this problem is to map instances to partitions and use these partitions to mitigate Input/Output (I/O) operation issues by aggregating communication and storage accesses.

A minimal configuration can be implemented to configure and test Netherite with DF, which can be done by editing the storageProvider parameter to Netherite in the host.json file in the Azure Storage account. Migration of TestHubNameHistory data is not supported and can not be done. Hence the existing applications will begin with clear TestHubNameHistory tables after moving to a new storage provider. [22]

The Durable Task Framework (DTFx) [23] is a distributed workflow execution engine that enables you to build long-running, stateful, and scalable workflows using the Microsoft .NET Framework. It is a library that lets developers write long workflows with monitoring and management operations in C# using simple async and await constructs.

Based on results from [6], Netherite performs and achieves better latency in all experiments. All in all, Netherite architecture improves the performance of implementing Durable function workflows.

2.2.3 Cloudburst

Cloudburst [24] is an attempt at redesigning serverless infrastructure to support consistent and fault-tolerant state management. Created by UC

Berkeley and Georgia Tech research teams, this stateful Function-as-a-Service (FaaS) offering overcomes several limitations of commercial FaaS systems.

It provides a Python API which helps write stateful application functions. These functions are scalable, fault-tolerant, and supported by Cloudburst's runtime. Like Apache Flink, Cloudburst also works with Directed Acyclic Graphs Directed Acyclic Graphs (DAG) to define the execution flow and dependencies between functions. These DAGs are generated by Cloudburst runtime based on user inputs. DAGs aid communication between functions. Anna [25] is a high-performing serverless key-value store used by Cloudburst to persist state. Anna provides capabilities such as auto-scaling and multi-tier serverless storage.

Cloudburst lets developers create and deploy stateful apps on top of serverless computing frameworks like AWS Lambda. Cloudburst offers consistent, scalable, and reliable data storage for stateful applications. As a result, Cloudburst can keep a state over several function calls, regardless of the success or failure of the underlying functions.

Cloudburst's additional features and capabilities include data partitioning, which enables developers to extend their applications horizontally by distributing data over different storage instances, and support for transactions, which enables developers to verify that state changes are atomic and consistent.

Overall, Cloudburst has been extensively accepted by programmers and academics across various disciplines and is for developers who wish to create and deploy stateful applications on different serverless systems.

2.2.4 Kalix

Kalix [26] is a Platform-as-a-Service (PaaS) offering that combines an API-first and a "database-less" programming model with a serverless runtime [26]

Kalix is a framework for building distributed applications using a microservices architecture. Each "monolithic" application is broken down into smaller services in a microservice architecture. These services are independent of each other and can be called in different parts of the application execution. This architecture also helps reuse services (functions) and eases debugging.

Like other stateful serverless systems discussed, Kalix handles all infrastructure-related responsibilities, such as setting up and tuning databases and maintaining and provisioning servers. Each service in Kalix is called a "module," consisting of a set of functions triggered by events. Kalix also provides compatibility with several programming languages to build functions. These functions are all executed using the Kalix runtime environment. A message-passing interface enables modules to communicate with one another, allowing them to share data and trigger events in other modules. As a result, programmers can construct complex applications out of smaller, independent components.

Kalix offers infrastructure and tools for setting up, scaling, and maintaining distributed applications. It also includes automated module scaling based on the running workload and support for deployments to different environments (such as the development stage, staging stage, and production stage). It helps developers troubleshoot and optimize their applications by logging and monitoring services.

Overall, Kalix aims to facilitate developer processes in creating and deploying distributed applications while providing the required tools and infrastructure to manage and scale them.

2.3 Sample application when built using a SSS

To explain how a stateful serverless system works, this sub-section highlights a sample stateful application such as shopping cart [27]. With this example, the code functioning is explained and how this application is different when built using a SSS instead of a traditional serverless system is explained as well.

Applications such as shopping carts and bank accounts represent complex logic representative of real-life scenarios. In these applications, we see that the functions can maintain state and message guarantees while the functions are invoked several times. In the shopping cart sample application, the egress showcases exactly-once processing.

A shopping cart sample application represents a web store where the shoppers or the users can add and remove items from their cart. Two functions do this - the StockFn function tracks the availability of the items in the web store stock, and the other function, UserShoppingCartFn, maintains the cart items of the shopper. Some tasks that the shopping cart application provides are adding and removing items from the shopper's cart, restocking items in the web store, helping with checkout, and sending a receipt at the checkout. These two functions are separate Java functions and the function service exposes these functions when exposed to an HTTP endpoint. In this case, the Flink StateFun runtime will manage the ingress, egresses, state and any communications between the functions. In the two functions there are several messages within which provide specific services such as RestockItem message is sent to the StockFn to restock an item and, AddToCart, Checkout and

ClearCart messages to the UserShoppingCartFn to add items to the cart, to checkout or clear the users cart respectively. Refer the sub-section 2.2.1 for how the system performs when a message is sent.

A shopping cart application built with DF uses the concept of an "Orchestrator function" to manage the state of each user's cart. The Orchestrator function would be triggered whenever a user takes an action that affects the state of their cart, such as adding an item or modifying the quantity of the item. The Orchestrator function would then execute one or more "activity functions" to perform the necessary tasks, such as updating the cart's state in a database or sending a notification to the user. Refer the sub-section 2.2.2 to further understand how the system works when a message is sent.

One way a shopping cart application built with DF would be different from one built with a traditional stateful serverless system is that the Orchestrator function would be able to manage the state of the cart across multiple activity functions. For example, consider a user adding an item to their cart and immediately modifying that item's quantity. In that case, the Orchestrator function could ensure that the correct quantity is stored in the cart by coordinating the execution of the two activity functions.

Another difference is that DF provide built-in support for features such as automatic retries and compensation, making building reliable and robust applications easier. Overall, a shopping cart application built with DF would be able to manage the state of each user's cart more flexibly and reliably than one built with a traditional serverless system.

One key difference between building this application using a stateful serverless system like Azure Flink StateFun versus using a traditional system is how the state is managed. The state is typically stored in a database or other persistent storage system in a traditional system, and updates are made directly to this storage.

In a stateful serverless system like Azure Flink StateFun, the state is managed by the stateful functions themselves. When a stateful function processes an event, it updates its local state, and this update is automatically persisted and made available to other stateful functions as needed. This means we do not have to worry about manually storing and retrieving states from a database, as the stateful serverless platform handles this for us.

There are several SSS applications with varying requirements. Generally, SSS needs a combination of different features, some of which have been discussed in this thesis. Specifically, having exactly-once processing, scalability / elastic scalability, being stateful, fully-managed runtime (serverless), communication patterns, and built-in primitives such as workflows and transactions are some features on which applications such as Shopping carts and Bank accounts depend.

2.4 Related work

There has been some work performed to compare systems. This thesis attempts to build comparison criteria that successfully compare systems architecturally, functionally, and on performance. The related work section highlights work that has helped and paved the way to choosing the appropriate approach while working and gaining some insights.

2.4.1 Comparison of FaaS

In the research paper "Function-as-a-Service: From an Application Developer's Perspective," Ali Raza from Boston University [28] explores the benefits and challenges of using FaaS from the perspective of application developers. The flexibility to concentrate on code development, lower operating expenses, and automated scalability are some of the primary advantages of FaaS mentioned in the study. The report does, however, also point out certain drawbacks of utilizing FaaS, including the requirement for new programming models and the potential for vendor lock-in scenarios.

The various types of available FaaS services, such as platform-specific and platform-agnostic options, are also covered, and their features and costs are evaluated. This research paper also discusses how some features and decisions made are one-time decisions and will affect the deployed systems and the user's use cases. Moreover, other online decisions can be updated, and the developer can be independent while choosing values for resources such as Central Processing Unit (CPU) and memory. On either type of decision, the performance and cost of the system are discussed.

The paper concludes that FaaS is a practical means for application developers. Still, weighing the available alternatives and considering the trade-offs concerning the application development needs and limitations is crucial before choosing one.

This paper raises the need to build comparison criteria when choosing one system over the other. This thesis incorporates the method by which comparisons were made, listing advantages, etc. This thesis takes different features into consideration. In particular, functionalities of the various SSS such as fault tolerance, local/in-memory state, and architectural features such as function patterns. Whereas, the paper [28] looks at features such as cold starts, reliability, and instance recycling times.

2.4.2 Performance Evaluation on Stateful Serverless

Benchmarking stateful serverless systems involves evaluating the performance of such systems under various workloads and conditions. One approach to benchmarking such systems is to use synthetic workloads that mimic real-world use cases and load patterns. The method, as mentioned earlier, allows system performance evaluation in a controlled and repeatable manner.

In this thesis, we run a system such as Flink StateFun [10] locally on a single machine. We use a test deployment to mimic the most acceptable way to build a system's proof-of-concept when choosing one system over the other based on its throughput performance.

In the research paper "Transactions across serverless functions leveraging stateful dataflows" by Martijn de Heus [1], the authors propose a new approach to building stateful serverless systems using dataflows. They argue that traditional approaches to building stateful serverless systems, such as databases or in-memory data stores, can be complex and costly. Instead, the authors propose using dataflows to enable stateful serverless functions to interact with shared states in a more efficient and scalable manner.

To evaluate their approach, the authors conducted a series of benchmarks using a synthetic workload designed to mimic a real-world use case. They compared the performance of their dataflow-based approach with that of a traditional database-based approach. They concluded that the dataflow-based approach had significantly lower latency and higher throughput.

Overall, the research in this paper [1] highlights the potential of dataflows to build efficient and scalable stateful serverless systems. Their approach offers a promising alternative to traditional approaches.

In [22], the evaluations of the execution engine are based on four workloads: Hello5, Word Count, Bank, and Collision Search. The hello5 workload is “A "hello world" workflow, each of which calls five tasks in sequence”. When run on a single core, this runs at 1 event per second for the original runtime. In this thesis, we are running a similar workload, the greeter function, which consists of an orchestration that calls one activity function and creates 4 events internally in the system based on the Azure Portal History table.

Chapter 3

Methodology

This chapter highlights the methodology undertaken to work on this thesis. The thesis is broken down into three review questions, namely Architectural qualities, Functional qualities, and Performance qualities. Each review question needs comprehension effort as reading and understanding different stateful serverless systems are very time-consuming. By the end of this chapter, we will be ready to work on features and technology with a plan on how the testbed will be created and how we can achieve results. Each section will prove the need for each Review Question.

3.1 Developing review questions

The thesis began with questions that needed to be answered when reading and learning about the stateful serverless system. Some of these questions are listed below:

- What are the architectural differences and similarities between each system? Process of gathering such information?
- What kind of application development is supported by the systems? Are there any concurrency limits? In-built functionality patterns? Fault-tolerance? Different kinds of state storage?
- What kind of in-built primitives are supported by the systems? For example, transactions, orchestrations, step-functions, and workflows.
- What is the process of migrating existing applications between these systems? How to migrate applications to serverless platforms?

- Are the systems language-agnostic? Does it affect performance?
- What is the performance of the systems? We should consider the impact of cold starts; elasticity; throughput; latency.

From the above questions, the thesis is bounded to three review questions which are mentioned as sub-goals for the thesis.

- Review Question I: **Architectural qualities**
- Review Question II: **Functionality qualities**
- Review Question III: **Performance benchmarking**

3.1.1 For Review Questions I: Architectural Qualities

Architectural design qualities are the backbone that supports all chosen stateful serverless systems, as all the features delivered by systems are backed and supported by the chosen architectural approach. The stateful serverless systems chosen have good official documentation and a varied list of features that can be compared and benchmarked.

There are several stateful serverless systems under constant development in the industry, but within the scope of this thesis, the stateful serverless systems taken into consideration are

- System 1: Flink StateFun [10]
- System 2: Durable Functions [20].
- System 3: Kalix [26]
- System 4: Cloudburst [24]

3.1.2 For Review Questions: II Functional Qualities

Functional qualities give purpose to the stateful serverless systems developed. Comparing and building a benchmark to understand functional qualities assist in choosing suitable stateful serverless systems to support specific use cases. To be capable of choosing one stateful serverless system over the other based on its functional qualities, it is necessary to truly understand features in-depth and be able to deploy and implement them based on use cases.

Functional qualities are the crux of every software and its implementation criterion. Within the scope of this thesis, the chosen stateful serverless systems to compare and benchmark are,

- System 1: Flink StateFun [10]
- System 2: Durable Functions [20].
- System 3: Kalix [26]
- System 4: Cloudburst [24]

And again, as this review question is heavily based on the available official documentation, the chosen stateful serverless systems have good official documentation with varied features that can be compared and analyzed.

3.1.3 For Review Question III: Performance Qualities

Several stateful serverless systems can be chosen for the performance benchmarking section of the thesis. Instead of populating this thesis with several of these systems, the two largely used and open-source available serverless systems are chosen,

- System 1: Flink StateFun [10]
- System 2: Durable Functions [20].

3.2 Methodology for Review Question-1

The approach to solving this review question is by reading the official documentation provided for each stateful serverless system. The official documentation is home to architectural design explanations, options for customized architecture implementations, and different ways of deploying a system.

By understanding the architectural qualities of stateful serverless systems, we can gauge how a chosen architecture affects the system's performance and feature development. Some questions that drive answers to this review questions are - Do certain architectural decisions narrow down possibilities of specific feature development in these systems? Are customization opportunities available to customize systems deployment to better suit different applications and their use cases?

Varied architectures bear varied implications which can affect feature support in the system. For example, Apache Flink StateFun [10] is built on top of a stateful streaming data flow engine. This architectural design affects how the stateful serverless systems handle state management, how key distributions are made, and how fault tolerance is maintained in the system [1]

A feature-by-feature comparison is performed while working on review question-1. As these official technical documentations are vast and complex, a feature-by-feature and qualitative approach helped achieve a good flow while working with different SSS and comparing them.

Choosing features that will be important when comparing SSS based on architectural qualities depends on the fundamental design on which each system is based. This approach also assists in building a sense of road-map for system developers and maintainers to prioritize which feature is evidently more required by their customers and why should a SSS be chosen over the other based on architecture and design.

3.3 Methodology for Review Question-2

Functional qualities help solution architects and infrastructure engineers build a customized SSS deployment that successfully supports application development and its use case.

Each application is developed based on user stories. Each user story develops into a critical feature that the stateful application should support. Similarly, while building a SSS that supports stateful application development, it is necessary to develop user stories that can assist in creating SSS features to maximize the adoption of the system. Hence, it is safe to pursue comparing functional qualities with the feature-by-feature qualitative approach.

The methodology used to compare and analyze existing functional qualities is by researching the latest valid product documentation. The product documentation for each of the chosen SSS is regularly updated but is overflowing with information. Technical documentation is developed by a team of technical content writers skilled in best explaining features and their deployments. These documentation articles are tested and approved by the respective SSS development and testing teams.

As we are heavily dependent on product documentation, our approach needs to take into account that product documentation is written with the intent to support existing customer cases and issues that have been raised internally. Hence, it is essential to steer into research-oriented pages and understand the crux of each feature without being limited by the official documentation which is created for the purpose of customer support and onboarding.

A list comprising each feature and its implementation technique in a generic form is challenging to achieve when supporting priority customers with customized needs through a single documentation. Hence, review question-2 is necessary when comparing multiple SSS.

The aim is to bridge the gap by creating a list of features for the comparison criteria that can aid in comparing and analyzing different SSS.

3.4 Methodology for Review Question-3

Successfully comparing systems, we need to figure out common grounds where the default deployments of these systems can be compared in simple terms. Benchmarking is also an excellent approach to understanding the process of the system and its performance, but that leads us to extend the use of a benchmark that is not built explicitly for SSS.
As these systems can have different state storage kinds, the write and read to the storage can be significantly improved by using one storage over the other. Keeping in mind the resource limitations with this thesis, we limit our tests to the default options and processes by the SSS for performance-based comparisons. The performance comparison is attempted with a quantitative approach.

3.4.1 Experimental designs and test cases

For performance comparisons, the workload for the evaluation is the Greeter Function. The Greeter function is a typical example used to demonstrate the concepts of Flink StateFun and DF. This function takes a name and user ID as input to keep count of every user and returns a customized greeting message. As it remembers information across multiple messages, this is stateful. This is a good sample application as it is easier to replicate when testing other new systems.

We ran the experiment by varying the number of events we sent to the system. The events are sent sequentially and blocked until the previous event has received a response/completed. The events have the same key. We repeated the execution 5 times and reported the averages thereof.

In Flink StateFun, the greeter application used in the thesis takes a name as input and returns a greeting message. It allows creating a greeting for a specified user name by sending a UserLogin message to the "userFn.java" in [29]. In DF, the greeter function is used as an activity function, which means it is called by the Orchestrator function to perform a specific task. The Orchestrator function triggers the Activity function with inputs of the "Name" in the greeting with the Activity trigger and uses the Azure default storage to maintain the state. If there is any failure, DF will power down and save the state. Once re-established, the Orchestrator function re-executes the entire function and helps rebuild the state with the help of DTFx. DTFx consults the execution history from the TestHubNameHistory table for the current orchestration.

In a stateful serverless system, **throughput** refers to the number of invocations or requests the system can process per second, minute, or hour. High throughput is generally desirable, indicating that the system can efficiently process many invocations. **Latency** can be defined as the time a stateful function takes to process an invocation and update and maintain the system's state. Low latency is generally desirable, as the system quickly responds to requests or process events. In this thesis, we use the latency parameter to investigate why a system performs poorly compared to others. In general, throughput and latency are inversely proportional to each other. But in some scenarios, latency and throughput might not be. For example, in some systems, increasing the number of tasks processed in parallel can increase latency due to the overhead of managing the parallel tasks and sharing resources.

For Flink StateFun:

- Test case: Run StateFun function in Java. Run 1000 sequential invocations to the function. Find total execution time and throughput.

For DF:

- Test case: Run Durable Function in C#. Run 1000 sequential invocations to the function. Find total execution time and throughput.

3.4.2 Test bed and software used

Running a stateful serverless system locally means running the system on a local machine rather than in a cloud environment. A stateful serverless system is a system that uses a serverless computing model in which the underlying infrastructure is abstracted away, and the system is designed to scale and manage the state automatically.

When running a stateful serverless system locally, the system is typically deployed and executed on a single machine rather than on a cluster of machines. This can be useful for development and testing, as it allows developers to iterate quickly and test the system without needing to deploy it to a cloud environment.

However, running a stateful serverless system locally can also have some limitations compared to running the system in a cloud environment. For example, the local machine may have different resources and scalability than a cloud environment, which can affect the performance and reliability of the system. The local machine has the following specifications - 4 CPU, Total memory available 12GB, instruction set - x86_64

Test bed creation for Apache Flink StateFun:

- Apache Flink StateFun Playground Git repository [29]
- Docker version 20.10.16, build aa7e414
- Docker compose version - docker-compose-1.29.2
- Use Client URL (cURL) commands to send events to the function

Test bed creation for Azure DF:

- To create and run a durable function in C#, the following prerequisites have to be fulfilled (along with versions used for this thesis) [30]
    - Install Visual Studio Code - 1.73.1
    - Extensions for using Visual Studio Code: Azure Functions and C#
    - Azure Functions Core Tools - 4.0.4895 (x64) is installed
    - Microsoft .NET Software Development Kit (SDK) 6.0.403 (x64) is installed
    - Azure storage subscription for students

Chapter 4

Implementation

This chapter explains the varied features considered while comparing different Stateful Serverless Systems. When choosing one system over the other, these features can be determined, affecting stateful application development and developer satisfaction. We arrive at our list of features and characteristics which help assess stateful serverless systems based on our developed review questions.

4.1 Review Questions 1 - Architectural Qualities

The work accomplished to understand how architectural qualities affect the system starts with understanding key features that affect successful stateful application development to compare and assess stateful serverless systems based on their architectural qualities.

When building a system, it is necessary to architect a solution that supports its users - developers, and researchers. In this case, to build and architect a stateful serverless system successfully, the goal is to build and design a reliable, scalable, secure, and cost-friendly stateful serverless system architecture.

Key features for review question-1 and how they can be defined in the world of stateful serverless systems:

- **Recovery Model**: As stateful serverless systems are introduced to maintain and manage the state of a stateful application. In case of any failure there needs to be a recovery model used to be able to recover from a failure state and the logic utilized is of importance.

- **Multiple function types**: Highlights the different kinds of function types available and how the function can be deployed to run in an application.
- **Scalability and auto-scaling capabilities**: If stateful serverless systems can dynamically support application development by scaling resources as required.
- **Compatibility to the different event triggering tools**: stateful serverless systems work with the logic of using event streaming tools to trigger function execution. As there are many such tools, the best-suited stateful serverless system will be capable of supporting most of them. This thesis compares with respect to three popular event streaming tools: Kafka, RabbitMQ, and Redis.
- **Function patterns or in-built application patterns**: Successful run of an application needs functions that need to be run either parallel or in sequential modes. This is called the chaining of functions.

The above-listed features can assist in reviewing the chosen stateful serverless systems with respect to their architectural and design qualities.

4.2 Review Question 2 - Functional Qualities

Successfully comparing systems based on their functionalities requires us to figure out different functionalities and features that will improve user satisfaction. These functionalities can be in-built with the system or a base for further customization that the user can achieve as per their needs. And, also helps us build user stories that explain the need for certain functions and their importance, proving why one stateful serverless system is better suited for the concerned stateful application development cycle.

The following list of qualities is proposed to compare all aspects of important functional qualities of stateful serverless systems. Key features for review question-2 and what they mean in the world of stateful serverless systems:

- **Built-in primitives**: Understanding how the system builds and uses segments of code that can be used to create more sophisticated code elements. Such as implementing workflows etc.

  - Actors: unit of computation that can receive, process, and send messages to other actors. They provide a way to build stateful, long-running, and fault-tolerant applications by exchanging messages and managing their private state.
  - Transactions: These are a way to ensure that a series of operations in a system are executed atomically and consistently. Transactions provide a way to ensure the state of a system is consistent and well-defined, even in the presence of failures or concurrent access, as they are either completed entirely or not executed at all.
  - Entities: unit of computation representing a shared state that multiple functions can access and modify. Entities are implemented as lightweight processes that can be deleted and scheduled independently.
  - Workflows: can be defined as the work in steps or tasks executed in a specific order in state serverless systems.

- **Fault tolerance**: The capability of a stateful serverless system to continue processing and working in case of component failure. Multiple types of failure tolerance can be implemented.

- **State or message guarantees provided**: Stateful application development requires the state to be stored and managed. Dive deeper by checking if the system uses a local or in-memory state logic.

- **Language-agnostic**: Assessing whether function development in different languages is supported. A system is language-agnostic when the functions and solutions do not depend on the programming language and specifications used. In any language, the function would perform and result in the same results, and there is no prejudice in using a specific language.

- **Concurrency - state**: In a stateful serverless system, managing concurrency refers to controlling how the system handles multiple requests or events simultaneously. This can be important because multiple requests or events may attempt to access and modify shared states simultaneously. If these requests or events are not correctly managed, it can lead to conflicts and inconsistencies in the application state. In most SSS, a request in line is not processed until the previous request is processed and that state has been updated.

The above-listed features can assist in reviewing the chosen stateful serverless systems with respect to their functional qualities.

4.3 Review Question 3 - Performance Qualities

This subsection showcases the test cases run, their results, issues hit, and what can be analyzed by these results.

4.3.1 Flink StateFun - Testcase 1

Using the playground example function [29], the system is deployed using docker-compose. Docker-compose builds the image for the code, and all necessary StateFun docker images and Apache Flink software are pulled. Use the docker-compose command to start running the services.

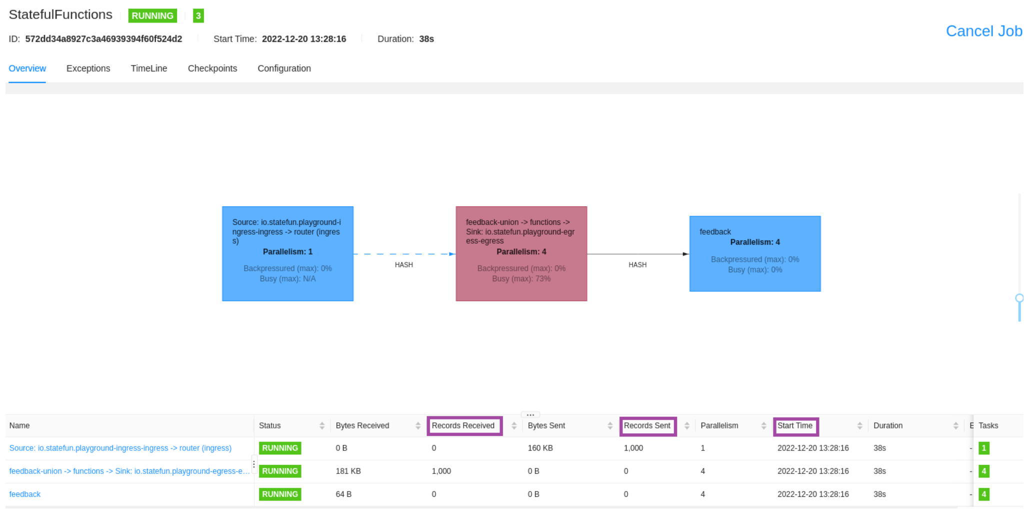
The greeter application is ready to process events that can be sent as cURL PUT request commands. As cURL commands can be run multiple times, this is an appropriate way to understand how StateFun manages the events.

Open another terminal to run the following to test the greeter function and check the Flink web dashboard for results on the event processing. As the system is run locally, check https://localhost:8081/#/overview to view the Apache Flink web dashboard. Under running Jobs, the greeter function number of records sent, records received, and feedback overview can be viewed. In all the invocations for Flink StateFun, we depend heavily on the dashboard and its live values of records received and sent. To view the messages sent to the egress, another cURL command can list the results from the egress; here, the GET request is used and would list the customized greetings for the user.

This thesis depends majorly on the StateFun dashboard provided, as it is challenging to timestamp the egresses and ingresses. When running a default deployment to create Proof-of-concept (POC) for a system, this thesis uses cURL to trigger the functions.

We calculate the time it takes for the system to send all the records. We use these timestamps from the Flink Stateful functions dashboard. The ingress and egress timestamps are recorded by noting when the cURL command was sent and when all the records were received. Figure 4.1 illustrates the system and the Flink stateful functions dashboard results when running 1000 messages sent to the Greeter Function.

We extend to test the system further by running the StateFun application and sending events using the cURL command. To send the events in 1000,


StatefulFunctions RUNNING 3

ID: 572dd34a8927c3a46939394f60f524d2 | Start Time: 2022-12-20 13:28:16 | Duration: 38s

Name	Status	Bytes Received	Records Received	Bytes Sent	Records Sent	Parallelism	Start Time	Duration	E	Tasks
Source: io.statefun.playground-ingress-ingress -> router (ingress)	RUNNING	0 B	0	160 KB	1,000	1	2022-12-20 13:28:16	38s	-	1
feedback-union -> functions -> Sink: io.statefun.playground-egress-e…	RUNNING	181 KB	1,000	0 B	0	4	2022-12-20 13:28:16	38s	-	4
feedback	RUNNING	64 B	0	0 B	0	4	2022-12-20 13:28:16	38s	-	4

Figure 4.1: Flink Stateful Functions dashboard

4000, 8000, and 16000 sequential invocations. We can find the execution time for each and the throughput as the number of invocations increases.

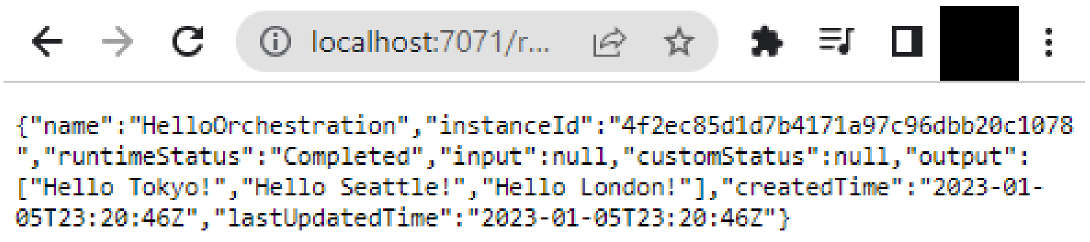
4.3.2 Durable Function - Testcase 1

To execute the activity function 1000 times, we write a simple for loop that triggers the function from the Orchestrator function sequentially. An Hypertext Transfer Protocol (HTTP) trigger function starts the experiment.

Using the Greeter function equivalent in the DF official tutorials to create a C# DF application. [30]

One way to calculate the execution time per invocation is by polling the status Universal Resource Identifier (URI) until the "runtimeStatus" parameter reaches the "Completed" state. The total execution duration is the difference between "lastUpdatedTime" from "createdTime". These details are from the JavaScript Object Notation (JSON) for the Orchestrator function status URI. Sample output to showcase how these parameters can be viewed and recorded is illustrated in figure 4.2.

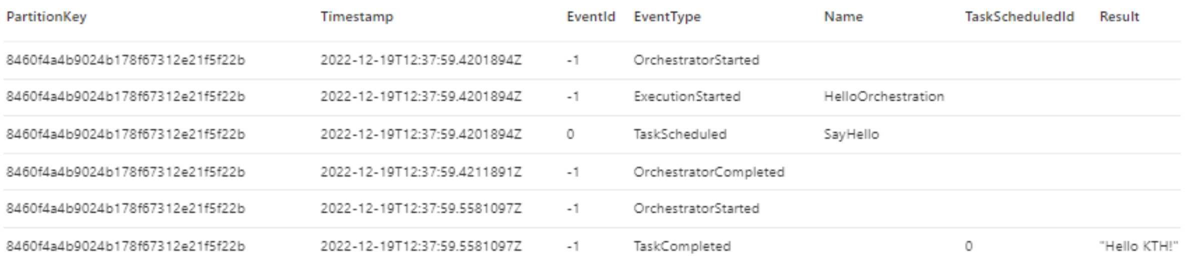
To know the total duration for running a large number of invocations, the timestamps from the first start of the Orchestrator function to the last event when the result is obtained, and the last Orchestrator function is completed need to be recorded. These details must be checked on the TestHubNameHistory Tables in the Azure Portal. Figure 4.3 is a sample table with entries.



[redacted]

```
{"name":"HelloOrchestration","instanceId":"4f2ec85d1d7b4171a97c96dbb20c1078
","runtimeStatus":"Completed","input":null,"customStatus":null,"output":
["Hello Tokyo!","Hello Seattle!","Hello London!"],"createdTime":"2023-01-
05T23:20:46Z","lastUpdatedTime":"2023-01-05T23:20:46Z"}
```

Figure 4.2: Sample DF status URI with JSON results



PartitionKey	Timestamp	EventId	EventType	Name	TaskScheduledId	Result
8460f4a4b9024b178f67312e21f5f22b	2022-12-19T12:37:59.4201894Z	-1	OrchestratorStarted			
8460f4a4b9024b178f67312e21f5f22b	2022-12-19T12:37:59.4201894Z	-1	ExecutionStarted	HelloOrchestration		
8460f4a4b9024b178f67312e21f5f22b	2022-12-19T12:37:59.4201894Z	0	TaskScheduled	SayHello		
8460f4a4b9024b178f67312e21f5f22b	2022-12-19T12:37:59.4211891Z	-1	OrchestratorCompleted			
8460f4a4b9024b178f67312e21f5f22b	2022-12-19T12:37:59.5581097Z	-1	OrchestratorStarted			
8460f4a4b9024b178f67312e21f5f22b	2022-12-19T12:37:59.5581097Z	-1	TaskCompleted		0	"Hello KTH!"

Figure 4.3: Sample Durable Function's TestHubNameHistory table

$$Throughput = \frac{(events\ per\ invocation) \times (number\ of\ invocation)}{total\ execution\ time} \tag{4.1}$$

Formula 4.2 explain how total execution time and throughput are recorded. Counting the number of events per invocation,

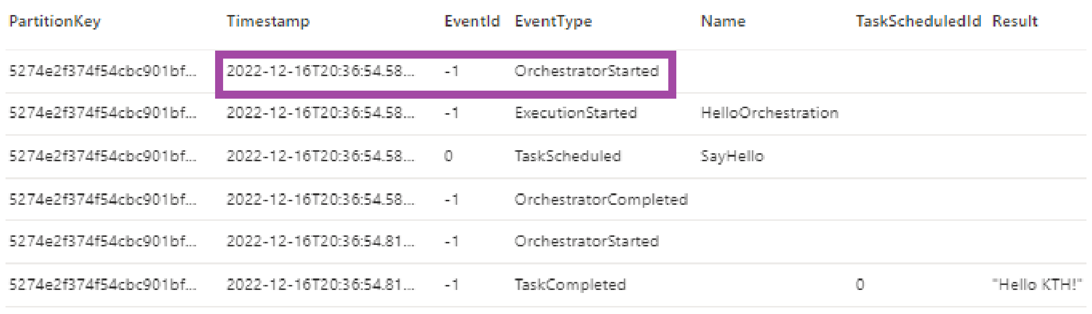
- Calling an Orchestrator function and handling the response is two events.
- Calling an activity function and handling its response is two events.

It is vital to count the number of events per invocation. In this case, events per invocation are 4.

The results per orchestration and the relevant timestamps can be viewed in the TestHubNameHistory tables in the Azure Storage account.

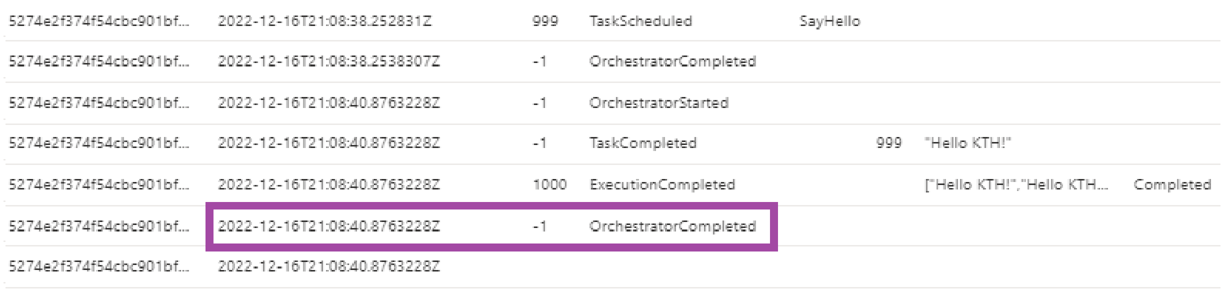
Figures 4.4 and 4.5 are TestHubNameHistory table entries with timestamps mentioned when the orchestration started and ended. These timestamps are highlighted.

Orchestrator function (Start timestamp) = 20:36:54
Orchestrator function (End timestamp) = 21:08:40



PartitionKey	Timestamp	EventId	EventType	Name	TaskScheduledId	Result
5274e2f374f54cbc901bf...	2022-12-16T20:36:54.58...	-1	OrchestratorStarted			
5274e2f374f54cbc901bf...	2022-12-16T20:36:54.58...	-1	ExecutionStarted	HelloOrchestration		
5274e2f374f54cbc901bf...	2022-12-16T20:36:54.58...	0	TaskScheduled	SayHello		
5274e2f374f54cbc901bf...	2022-12-16T20:36:54.58...	-1	OrchestratorCompleted			
5274e2f374f54cbc901bf...	2022-12-16T20:36:54.81...	-1	OrchestratorStarted			
5274e2f374f54cbc901bf...	2022-12-16T20:36:54.81...	-1	TaskCompleted		0	"Hello KTH!"

Figure 4.4: Durable Functions: TestHubNameHistory table entry of Orchestrator start



5274e2f374f54cbc901bf...	2022-12-16T21:08:38.252831Z	999	TaskScheduled	SayHello			
5274e2f374f54cbc901bf...	2022-12-16T21:08:38.2538307Z	-1	OrchestratorCompleted				
5274e2f374f54cbc901bf...	2022-12-16T21:08:40.8763228Z	-1	OrchestratorStarted				
5274e2f374f54cbc901bf...	2022-12-16T21:08:40.8763228Z	-1	TaskCompleted		999	"Hello KTH!"	
5274e2f374f54cbc901bf...	2022-12-16T21:08:40.8763228Z	1000	ExecutionCompleted			["Hello KTH!","Hello KTH...	Completed
5274e2f374f54cbc901bf...	2022-12-16T21:08:40.8763228Z	-1	OrchestratorCompleted				
5274e2f374f54cbc901bf...	2022-12-16T21:08:40.8763228Z						

Figure 4.5: Durable Functions: TestHubNameHistory table entry of 1000th event completion

The total execution time = 21:08:40 - 20:36:54
Total execution time = 00:31:46 = 1906 seconds

Throughput calculated is 2 events per second approximately when running locally.

Chapter 5

Results and Assessments

This chapter showcases the results of comparisons performed and is divided into three main sections based on the review questions derived. Each section has a results sub-section followed by a result assessment sub-section.

5.1 Review question-1 Architectural Qualities

The following table lists the comparison criteria taken into consideration when comparing chosen stateful serverless systems based on architecture.

Table 5.1: Results for Review question - 1

Feature | System	StateFun	DF	CloudBurst	Kalix
Recovery model	✓[31]	✓[7] [23]	✓[24]	✓[26]
Multiple Function types	✓[10]	✓[7]	✗	✗
Scalability-auto-scaling	✓[10]	✓[32]	✓[24]	✓[26]
Event streaming - Kafka	✓[33]	✓[34]	✓[2]	✓[35]
Event streaming - RabbitMQ	✓[10]	✗	✗	✗
Event streaming - Redis	✓[8]	✗	✓[24]	✗
In-built application patterns	✗	✓[7]	✗	✗

✓ - Supported, [✗] - Not supported, [NS] - Not Stated

5.1.1 Assessment for Review question-1

The architectural comparison result table shows that essential and necessary features can be considered further. Stateful serverless systems are designed to have a scalable and highly available recovery model because a robust cloud infrastructure of high availability, multiple zones, etc., supports it. Being fault tolerant with a recovery model, scalable, and highly available are the main reasons a stateful application development is best suited to SSS. We have considered these parameters to understand if these features are implemented in the system. To create comparison criteria, we have chosen parameters and features which are either available or not available. Features such as function patterns, having system-provided function types, etc., support the decision-making process of choosing a stateful serverless system. These features help select the best-suited system according to the stateful application and the application development team's requirements.

Multiple function types - In systems such as Flink StateFun and DF, there are system-defined functions such as the Orchestrator function in DF. Based on architecture, this aids in the easier adoption of such systems.

Function patterns - In systems such as Azure Durable, application patterns are available. Architecturally this aids in designing stateful applications where different patterns and workflows are needed. It also makes it more effortless for developers to implement complex applications. [36]

According to the Table 5.1, If the application can be benefited from using different function patterns, DF would be a suitable SSS to select. If not, then both StateFun and DF would both be suitable SSS.

5.2 Review question-2 Functional Qualities

The following table lists the comparison criteria taken into consideration when comparing chosen stateful serverless systems based on functionality,

✓ - Supported, [✗] - Not supported, [NS] - Not Stated

5.2.1 Assessment for Review question-2

The functional comparison result table shows that essential features can be considered further when decision-making. Features such as fault tolerance, auto-scalability, and concurrency are available in all the chosen systems. Fault tolerance is an essential criterion when judging a SSS, but in different ways, all SSS provide fault tolerance. To create comparison criteria, we have chosen

Table 5.2: Results for Review question - 2

Feature | System	StateFun	DF	CloudBurst	Kalix
Fault tolerance	✓[37]	✓[20]	✓[24][25]	✓[26]
State/Message guarantees-Database	✗	✓[38]	✗	✗
State/Message guarantees-Snapshot	✓[31]	✗	✗	✗
Local/ in-memory state	✓[10]	✓[30]	✗	✓[26]
Language agnostic	✓[39]	✓[39]	✓[24]	✗ [40]
Concurrency - state updates	✗	✗	✓[24][25]	✗
In-built primitive - Actors or Actions	✗	✗	✗	✓[41]
In-built primitive - Transactions	✓[1]	✗	✗	✗
In-built primitive - Entities	✗	✓[42]	✗	✓[43]
In-built primitive - Workflows	✗	✓[7]	✗	✗

parameters and features which are either available or not available. We have taken state and message guarantees further by comparing how the state is managed. We also help get a brief idea of how the SSS provided recovery model (from review question-1) is implemented. It signifies how architecture and functional features are linked in the SSS context. Features such as in-memory or local state management, being language agnostic, and having support for built-in primitives improve the overall judgment of the system and make for a successful comparison concerning functionality.

In-memory/ local state management - Highlights if the system stores the function state in-memory as this makes the system respond to calls faster. Systems such as Apache Flink StateFun [10] have a local state accessed during the function calls.

Language-agnostic - Being language agnostic means that the system supports all kinds of development languages, and there is no prejudice in choosing a specific language due to system limitations.

In-built primitives - Built-in primitives refer to basic functionality or features provided by a stateful serverless system out-of-the-box, without the need for additional configuration or development. These primitives can be necessary when comparing and working with stateful serverless systems because they can significantly affect the system's ease of use, performance, and cost. Moreover, suppose a stateful serverless system lacks specific built-in primitives. In that case, it then requires more customization and development to meet the application's needs, increasing the complexity and cost of using the system.

Based on Table 5.2, essential features are in-built primitives and local/in-memory states. StateFun and DF would be good options for applications that require these features.

5.3 Review question-3 Performance Qualities

5.3.1 Flink StateFun

Flink StateFun is an event-driven system. Hence, after running the greeter application, we send messages as events to the function. To test this system, we send 1000 invocations sequentially using the cURL command.

To test the system further, we run sequential events to invoke the function in 1000, 2000, 3000, and 4000. As we did not see a difference in throughput, we decided to run messages in larger numbers. Therefore, table 5.3 showcases throughput for 1000, 4000, 8000, and 16000 invocations. Figure 5.1 showcases the throughput values recorded graphically.

Table 5.3: StateFun - Throughput results for varying number of invocations

Number of Invocations	Execution time(sec)	Throughput approx(events/sec)
1000	36	28
4000	83	48
8000	127	63
16000	246	65

The graph shows a gradual increase in throughput as the number of invocations increased. For this thesis, the Flink StateFun testbed has 1 Task Manager and 4 Task slots. A default parallelism of 4 is set. One possible reason why the throughput is increasing could be that parallelism is being functioned, thereby more invocations are processed and hence, better throughput values.

5.3.2 Durable Functions

The stateful application is executed for testing Durable Functions, and the function is invoked 1000 times. As part of a single invocation, the Orchestrator function triggers the Activity function 1000 times. For the test case running

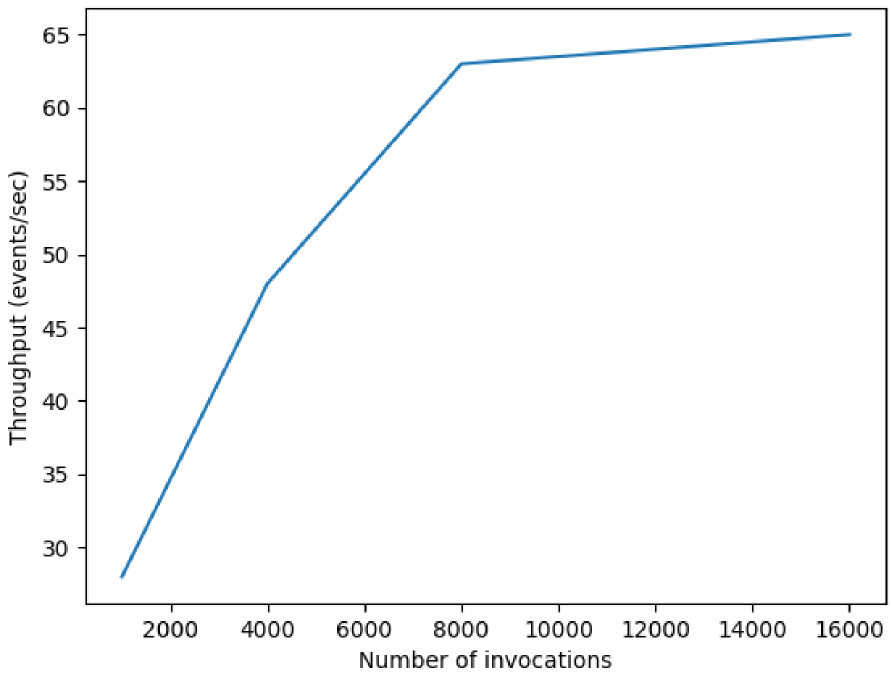


Figure 5.1: StateFun Throughput graph with varying number of invocations

1000 sequential invocations, the throughput result is approximately 2 events per second.

To test further, we execute the application with different numbers of invocations to calculate throughput. We test further by executing the application with 1000, 2000, 3000, and 4000 sequential invocations.

In the case of DF, the total number of events is equal to the product of the number of invocations and the number of events per invocation. Hence, When running 4000 sequential invocations, the system has 16000 events. Because each invocation creates 4 events, as described earlier. Table 5.4 showcases the throughput results in a tabular form. Figure 5.2 showcases the throughput value recorded graphically.

5.3.3 Assessment for review question-3

In the tests conducted, we can observe that Flink StateFun has more promising results than DF concerning Performance in throughput and execution time

Table 5.4: DF - Throughput results for varying number of invocations

Number of invocations	Execution Time (sec)	Throughput (events/sec)
1000	1906	2.1
2000	4470	1.78
3000	10132	1.2
4000	18023	0.8

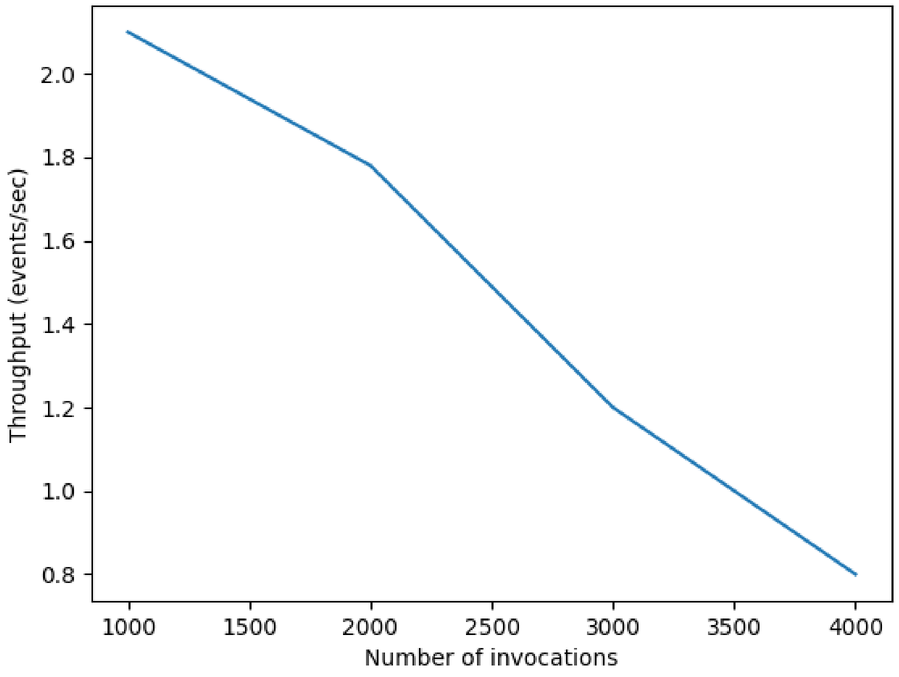


Figure 5.2: DF- Throughput graph with varying number of invocations

when running invocations sequentially.

Flink StateFun resulted in 28 events/second as throughput with 1000 invocations. We executed 4000, 8000, and 16000 sequential invocations to test further. It resulted in 48, 63, and 68 events/second as throughput, respectively. DF resulted in approximately 2 events/second as throughput when running 1000 sequential invocations. This throughput decreased to approximately 1 events/second after increasing the number of invocations.

Compared to Flink StateFun, we see that the throughput of DF is much lower. To investigate, we plot the average latency of these executions to analyze the system performance. The average latency of the system is calculated by finding the difference between each ingress timestamp and its corresponding egress timestamp from the Azure Portal TestHubNameHistory table. These values are then used to calculate the latency averages plotted in Figure 5.3. It is seen that there is a gradual increase in latency (in seconds) with the increase in the number of invocations. This indicates that the function takes longer to execute as the workload increases. Overall, the throughput values for DF decreases, and the average latency (in seconds) increases with an increase in the number of invocations.

There could be some possible reasons why DF performs. One reason could be that the default parallelism of 4 is insufficient, as the number of invocations and the total number of events that get executed are large with Azure Storage Provider. Another reason could be computational resources such as CPU and I/O.

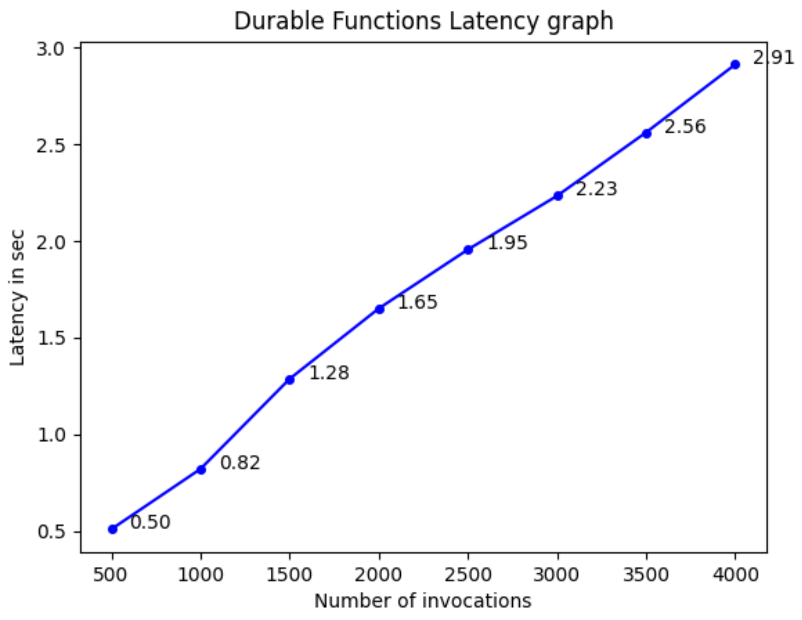


Figure 5.3: Durable Functions: Average latency graph

The results lead us to infer that in DF, the application needs a different programming model implemented and different application patterns to customize how the activity function can be run to achieve better performance metrics. Components such as sub-orchestrators can be utilized. With an increase in computational resources, the throughput results can be improved in the case of sequential executions.

Apache Flink StateFun performs better when executing a stateful application such as the Greeter application while invoking sequentially. DF are designed to allow a single invocation to trigger multiple internal function triggers and create events, which can help implement complex stateful logic. In this case, 4 events per invocation. In contrast, Apache Flink StateFun is designed to process each event or message to a stateful function as a separate task, with no internal events created or triggers.

Chapter 6

Discussion

This chapter highlights points in the thesis which raise concerns and are good points to discuss further and, thereby, add significance to the thesis results.

In the test case for recording throughput for DF, it is seen that each task or invocation runs 4 events (this can change depending on the stateful application). Hence, executing 1000 invocations makes 4000 events in the system, and executing 4000 invocations makes 16000 events. In Apache Flink StateFun, each task (message to the function) is a single message to the function and does not internally trigger multiple functions triggers to complete one invocation like in the case of DF. Hence, further testing of Flink StateFun is limited to running 16000 invocations.

Several factors affect the throughput of a system. The number of events or messages processed by the system is just one-factor affecting throughput. Many other factors also impact throughput, such as the complexity of the event processing logic, the amount of state that needs to be managed, and the available resources (e.g., CPU, memory, network bandwidth). To accurately compare the throughput of DF and Apache Flink StateFun, it would be necessary to consider these and other factors. In this thesis, we have tried to maintain similar scenarios while comparing both systems.

Choosing the proper application which can be used to test different stateful serverless systems was a difficult task. In these test cases, the stateful application must create a good sample application that can be easily implemented for new stateful serverless systems. The greeter Function fits the bill perfectly and could be easily incorporated for both the systems chosen for performance comparisons.

The reported throughput was approximately 2 events per second for DF and 28 events per second for Flink StateFun with 1000 sequential invocations. Compared to previous studies, the DF throughput is similar; when we compare the greeter function to the Hello5 function in [22], we reported 2 events/second for 1000 invocations, whereas they reported 1 event per second, as the workloads are similar. Compared to Flink StateFun, the only evaluation we have found is from [1], which reports around 20 thousand events per second for a workload that runs transactions on a modified benchmark based on Yahoo! Cloud Serving Benchmark (YCSB). Our results are low in comparison, this could be because our workload blocks and waits to send the next event until the response for the previous event has been received. Thus, we do not utilize the underlying potential for parallelism. In addition, the experiments were evaluated locally, whereas they were run on a distributed cluster in the other work. A limitation of our workload is that it may not represent real stateful serverless applications. In comparison to the shopping cart, for example, our workload handles less state, and the interaction pattern is less complex.

Some trade-offs need to be addressed,

- Between feature-rich and performance: Adding new interesting features can improve the overall use case of the SSS, but it can also cause lower performance metrics of the SSS. These trade-offs while building systems with respect to functionality and design can dictate a SSS performs with the growing demands of the stateful application. It is also possible that some features can create limitations to the performance.
- Based on features: As not every SSS has all features taken into consideration, if you choose one SSS, you might get some features, whereas other features may not be supported for this SSS. For example, if you need workflows, you can choose DF. But if you also need transactions, you can choose between DF and StateFun, as neither has both feature.

Based on our results, we would make the following suggestions for developers on their choice. These suggestions can work as a set of guidelines that Product Owners and developers can use when selecting a SSS.

- Based on Architectural comparisons in review question-1, the suggestion is to use Azure DF if the stateful application is dependent on using function patterns.
- Based on Architectural comparisons in review question-1, if the stateful

application can benefit from using different kinds of function types, the suggestion is to use DF and Apache Flink StateFun.

- Based on Functional comparisons in review question-2 if the stateful application can benefit from using local or in-memory state. The suggestion is to use StateFun, DF, and Kalix.
- Based on Functional comparisons in review question-2 if the stateful application can benefit from using built-in primitives. If using Actors, then Kalix is the suggested SSS. If using transactions as a built-in primitive, then StateFun is the suggested SSS. In the case of Entities as a built-in primitive, using DF and Kalix is suggested. Similarly, for using workflows, only DF is suggested SSS.
- Based on Performance comparisons in review question-3 between StateFun and DF, to support critical stateful applications based on the throughput of the system. The suggested SSS is Apache Flink StateFun. (Although with newer versions (Netherite), DF might also provide good performance.)

Chapter 7

Conclusions and Future work

This chapter talks about the conclusions that can be drawn from the experiments performed to answer each review question. This section also includes the future work and limitations, which are not included in this thesis due to time constraints but are of importance to the goal of this thesis.

7.1 Conclusions

Comparing stateful serverless systems should consider multiple factors, including architecture, functionality, and performance. This thesis is scoped to work within these three aspects.

In architecture, a system's different function types and patterns can impact performance and, when used correctly, result in better throughput values. For example, in the case of DF, using a different function pattern, such as fan-in/fan-out, might help improve the system performance as it does multiple parallel activity execution. The Orchestrator function executes many activity functions in parallel.

We can make suggestions based on the thesis results and provide a guideline when focusing on architectural comparisons. We conclude that DF would be a good fit for using function patterns when developing complex stateful applications. And, StateFun and DF when multiple function types benefit the application.

Functionality is an essential element to consider when comparing stateful serverless systems. State management, such as local and in-memory state, is a key feature that can impact the system's ability to handle stateful workloads. Language-agnostic systems can offer more flexibility in terms of the programming languages used. In-built primitives make it easier to develop

complex applications on the system.
We conclude by making a guideline that StateFun, DF, and Kalix are good SSS to consider. But when we add other features, this can lead to feature combinations that are supported as not all systems support all features taken into consideration. Such as when using built-in primitives like Actors, Entities, Workflows, etc.

Lastly, performance is a significant factor when comparing stateful serverless systems. The conclusion and guideline is that systems such as Azure DF may require an application programming model that incorporates different function patterns and parameters to perform at par with SSS, such as Apache Flink StateFun, in the case of sequential function executions. However, Durable Functions has other advantages, such as better connectivity to external services through the Azure Portal, which could make them more appealing for specific applications.

With the thesis test cases and corresponding results, we can suggest that Apache Flink StateFun is a better-suited SSS as it has better throughput than Azure DF when testing with a sample example such as the Greeter Function and using sequential invocations to the function.

In this attempt to compare different stateful serverless systems based on three factors, the insight we gain from completing this thesis is that the application and function patterns and features, such as in-memory state management used to develop an application, bring about a big difference in the system's performance which is crucial to its success.

7.1.1 Trade-offs

As mentioned earlier, we can conclude by mentioning certain trade-offs when comparing SSS.

- Between SSS being feature-rich and the performance of the SSS: Adding more functionality and being feature-rich can be a big reason for the easy adoption of a SSS. But, these can also cause lower performance metrics by the SSS. This could be why DF performs lower throughput values than StateFun for sequential function invocations, as DF is a good example of a feature-rich SSS.

- Based on features: Based on the features taken into consideration, all features are not available in each of the chosen SSS. This leads us to recognize and conclude that when choosing a SSS, a user might get some features, and some features may not be supported. Hence, the choice should be made by understanding the needs of the stateful application.

7.2 Limitations

The thesis is scoped to compare the systems based on architecture, functionality, and performance. There are many factors to which this comparison can be extended. Factors such as,

- Cost: The cost of using a stateful serverless system can be an important factor to consider, particularly for applications that are expected to have high usage or that will be running for an extended period of time.
- Security: The security measures in place for a stateful serverless system can be an important factor to consider, as it can impact the system's ability to protect data and maintain confidentiality.
- Community support: The level of community support for a stateful serverless system can be an important factor to consider, as it can impact the availability of documentation, examples, and assistance when working with the system.

7.3 Future work

Future research needs to be conducted to compare and evaluate stateful serverless systems, not just based on their architecture, functionality, and performance. The following lists some of these factors that can be considered for further work,

- Functionality comparisons: It would be helpful to conduct more detailed comparisons of the functionality offered by different stateful serverless systems, which can involve evaluating the systems' capabilities in different scenarios and use cases.
- Evaluation of application programming models: It would be helpful to conduct more in-depth evaluations of the application programming models used by different stateful serverless systems, including the types of patterns and parameters available and how they impact performance and functionality. To extend the evaluation through more realistic workloads, for example, the shopping cart example. Re-evaluate the

systems using the shopping cart example, and aim to achieve higher throughput. The throughput should be comparable to previous related work mentioned in the Background section.

- Cost analysis: It would be helpful to conduct more comprehensive analyses of the costs associated with using different stateful serverless systems, including upfront and ongoing costs. This could involve considering factors such as usage patterns, scale, and integration with other services.
- Testing the systems on different cloud resources: Evaluating how the systems perform when running on different Cloud platforms. For example, comparing the systems on different cloud providers (such as Azure, Amazon Web Services, or Google Cloud Platform) with different regions or locations within a single provider. This could help to determine how the systems are affected by factors such as data center infrastructure and other platform-specific differences.

References

[1] M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos, "Transactions across serverless functions leveraging stateful dataflows," *Information Systems*, vol. 108, p. 102015, 2022. doi: https://doi.org/10.1016/j.is.2022.102015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437922000229 [Pages 1, 4, 18, 22, 37, and 44.]

[2] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *CoRR*, vol. abs/1902.03383, 2019. [Online]. Available: http://arxiv.org/abs/1902.03383 [Pages 2 and 35.]

[3] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018. ISBN 9781931971447 p. 133–145. [Page 2.]

[4] A. Grumuldis, "Evaluation of "serverless" application programming model: How and when to start serverles," Ph.D. dissertation, 2019. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-247625 [Page 2.]

[5] "Aws lambda documentation," 2022, accessed on June 4, 2022. [Online]. Available: https://aws.amazon.com/lambda/ [Page 2.]

[6] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Serverless workflows with durable functions and netherite," *CoRR*, vol. abs/2103.00033, 2021. [Online]. Available: https://arxiv.org/abs/2103.00033 [Pages 2, 3, 12, and 13.]

[7] "What are durable functions?" 2022, accessed on June 4, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp [Pages 2, 3, 11, 35, and 37.]

[8] "Apache flink documentation," 2022, accessed on June 4, 2022. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.14/#apache-flink-documentation [Pages 2 and 35.]

[9] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, "Stateful serverless computing with crucial," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, mar 2022. doi: 10.1145/3490386. [Online]. Available: https://doi.org/10.1145/3490386 [Pages 3 and 9.]

[10] "Stateful functions: A platform-independent stateful serverless stack," 2022, accessed on June 4, 2022. [Online]. Available: https://nightlies.apache.org/flink/flink-statefun-docs-release-3.0/ [Pages 3, 9, 18, 20, 21, 35, and 37.]

[11] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," *CoRR*, vol. abs/2012.14132, 2020. [Online]. Available: https://arxiv.org/abs/2012.14132 [Page 3.]

[12] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," 2017. [Online]. Available: https://arxiv.org/abs/1706.03178 [Pages 8 and 9.]

[13] W. Zhang, V. Fang, A. Panda, and S. Shenker, "Kappa: A programming framework for serverless computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3419111.3421277. ISBN 9781450381376 p. 328–343. [Online]. Available: https://doi.org/10.1145/3419111.3421277 [Page 9.]

[14] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'20. USA: USENIX Association, 2020. ISBN 978-1-939133-19-9 [Page 9.]

[15] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021. doi: 10.1145/3477132.3483541. ISBN 9781450387095 p. 691–707. [Online]. Available: https://doi.org/10.1145/3477132.3483541 [Page 9.]

[16] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. Stoica, "Jiffy: Elastic far-memory for stateful serverless analytics," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022. doi: 10.1145/3492321.3527539. ISBN 9781450391627 p. 697–713. [Online]. Available: https://doi.org/10.1145/3492321.3527539 [Page 9.]

[17] "Ibm cloud functions," 2022, accessed on May 20, 2022. [Online]. Available: https://cloud.ibm.com/functions/ [Page 9.]

[18] "Google cloud functions," 2022, accessed on May 19, 2022. [Online]. Available: https://cloud.google.com/functions [Page 9.]

[19] T.-L. G. Tai, "Stateful functions internals: Behind the scenes of stateful serverless," 2022, accessed on June 4, 2022. [Online]. Available: https://flink.apache.org/news/2020/10/13/stateful-serverless-internals.html [Pages ix, 9, and 10.]

[20] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Durable functions: Semantics for stateful serverless," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. doi: 10.1145/3485510. [Online]. Available: https://doi.org/10.1145/3485510 [Pages 11, 20, 21, and 37.]

[21] "Task hubs in durable functions (azure functions)," 2022, accessed on October 12, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-task-hubs?tabs=csharp [Pages ix and 12.]

[22] "Azure durable functions: Supported languages," 2022, accessed on June 4, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/quickstart-netherite [Pages 13, 18, and 44.]

[23] "What are durable functions -the technology," 2022, accessed on June 4, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp#the-technology [Pages 13 and 35.]

[24] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2438–2452, sep 2020. doi: 10.14778/3407790.3407836. [Online]. Available: https://doi.org/10.14778/3407790.3407836 [Pages 13, 20, 21, 35, and 37.]

[25] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein, "Anna: A kvs for any scale," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018. doi: 10.1109/ICDE.2018.00044 pp. 401–412. [Pages 14 and 37.]

[26] "Kalix: Official documentation," 2021, accessed on April 19, 2022. [Online]. Available: https://docs.kalix.io/ [Pages 14, 20, 21, 35, and 37.]

[27] T. A. S. Foundation, "apache/flink-statefun-playground: Shopping cart function," 2022, accessed on June 4, 2022. [Online]. Available: https://github.com/apache/flink-statefun-playground/tree/main/java/shopping-cart [Page 15.]

[28] A. Raza, I. Matta, N. Akhtar, V. Kalavri, and V. Isahagian, "Sok: Function-as-a-service: From an application developer's perspective," *Journal of Systems Research*, vol. 1, 09 2021. doi: 10.5070/SR31154815 [Pages 17 and 18.]

[29] T. A. S. Foundation, "apache/flink-statefun-playground: Java greeter function," 2022, accessed on June 4, 2022. [Online]. Available: https://github.com/apache/flink-statefun-playground/tree/main/java/greeter [Pages 23, 25, and 30.]

[30] "Durable functions: Create your first durable function in c," 2022, accessed on June 4, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-create-first-csharp?pivots=code-editor-vscode [Pages 25, 31, and 37.]

[31] "Fault tolerance via state snapshots," 2022, accessed on November 27, 2022. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/learn-flink/fault_tolerance/ [Pages 35 and 37.]

[32] "Performance and scale in durable functions, auto-scaling," 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-perf-and-scale [Page 35.]

[33] "Apache kafka," 2022, accessed on August 7, 2022. [Online]. Available: https://nightlies.apache.org/flink/flink-statefun-docs-master/docs/modules/io/apache-kafka/ [Page 35.]

[34] "Apache kafka bindings for azure functions overview," 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-kafka?tabs=in-process2Cportal&pivots=programming-language-csharp [Page 35.]

[35] "Kalix, configure message brokers," 2022, accessed on June 28, 2022. [Online]. Available: https://docs.kalix.io/projects/message-brokers.html [Page 35.]

[36] "Azure durable functions: Application patterns," 2022, accessed on June 4, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp#application-patterns [Page 36.]

[37] "Application building blocks, fault tolerance," 2022, accessed on May 23, 2022. [Online]. Available: https://nightlies.apache.org/flink/flink-statefun-docs-release-3.0/docs/concepts/application-building-blocks/#fault-tolerance [Page 37.]

[38] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "Faster: A concurrent key-value store with in-place updates," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018. doi: 10.1145/3183713.3196898. ISBN 9781450347037 p. 275–290. [Online]. Available: https://doi.org/10.1145/3183713.3196898 [Page 37.]

[39] "Azure durable functions: Supported languages," 2022, accessed on June 4, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/supported-languages [Page 37.]

[40] "Kalix: Supported languages," 2021, accessed on June 4, 2022. [Online]. Available: https://docs.kalix.io/reference/supported-languages.html [Page 37.]

[41] "Kalix process overview, actions," 2022, accessed on October 12, 2022. [Online]. Available: https://docs.kalix.io/developing/development-process-js.html#_actions [Page 37.]

[42] "Stateful entities, entity functions," 2022, accessed on October 12, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities?tabs=csharp [Page 37.]

[43] "Kalix process overview, stateful entities," 2022, accessed on October 12, 2022. [Online]. Available: https://docs.kalix.io/developing/development-process-js.html#_actions [Page 37.]

TRITA-EECS-EX-2023:20