Degree Project in Computer Science and Engineering, specialising in Communication Systems

Second cycle, 30 credits

Stateful Serverless Computing: A Comparative Analysis of Orchestrator-based and Actor-like Models

FURKAN ÜN

Stateful Serverless Computing: A Comparative Analysis of Orchestrator-based and Actor-like Models

FURKAN ÜN

Master's Programme, Communication Systems, 120 credits
Date: July 29, 2025

Supervisors: Matti Siekkinen, Ki Won Sung
Examiner: Slimane Ben Slimane
School of Electrical Engineering and Computer Science
Swedish title: Tillståndsbaserad serverlös databehandling: En jämförande analys av orchestrator-baserad och actor-liknande modell

© 2025 Furkan Ün

Abstract

Serverless cloud computing, also mostly known as Function-as-a-Service (FaaS) is a popular cloud model where cloud users write a piece of code, or, namely, functions, and cloud providers take the responsibility of the underlying operational aspects, such as execution, auto-scaling, and monitoring of these functions, instead of the cloud users themselves. While this model provides benefits and is widely adopted, the stateless nature of FaaS led to the evolution of initial serverless functions to more advanced and architectural patterns that we refer to as a stateful serverless system (SSS) in this thesis. SSS provides abstractions that hide the challenges of managing and persisting state while keeping the benefits of serverless functions, such as auto-scaling and usage-based billing. In this thesis, we conducted a comparative analysis of stateful serverless architectures, where we specifically focused on two different architectural approaches: orchestrator-based and actor-like models. For this purpose, we chose Azure Durable Functions (ADF) and Apache Flink Statefun (FSF) to compare their architectural differences and performance characteristics, such as latency and throughput, through quantitative benchmarking. In our benchmarking scenarios, FSF consistently achieved the highest throughput and lowest latency in all tested experiments, while ADF with the Default backend configuration showed modest performance due to its I/O bounded state access pattern. Our benchmarking results provide quantitative metrics about the performance characteristics and architectural attributes of stateful serverless approaches, contributing to the understanding of these systems. The results and contributions of this thesis are valuable for system architects, developers, and researchers.

Keywords

Serverless computing, Stateful Serverless, Azure Durable Functions, Apache Flink Statefun, Function-as-a-Service, State Management

Sammanfattning

Serverlös molnbaserad databehandling, även mest känd som Function-as-a-Service (FaaS), är en populär molnmodell där molnanvändare skriver kod i form av funktioner, och molnleverantörer tar ansvar för de underliggande operativa aspekterna, såsom exekvering, automatisk skalning och övervakning av dessa funktioner, istället för att användarna själva gör det. Även om denna modell erbjuder fördelar och är brett adopterad, har den tillståndslösa naturen hos FaaS lett till en utveckling från initiala serverlösa funktioner till mer avancerade arkitekturmönster som i denna avhandling benämns som ett tillståndsbaserat serverlöst system (SSS). SSS tillhandahåller abstraktioner som döljer utmaningarna med att hantera och bevara tillstånd, samtidigt som fördelarna med serverlösa funktioner, såsom automatisk skalning och användningsbaserad debitering, bibehålls. I denna avhandling har vi genomfört en jämförande analys av tillståndsbaserade serverlösa arkitekturer, där vi specifikt fokuserat på två olika arkitektoniska tillvägagångssätt: orchestrator-baserade och actor-liknande modeller. För detta ändamål valde vi Azure Durable Functions (ADF) och Apache Flink Statefun (FSF) för att jämföra deras arkitektoniska skillnader och prestandaegenskaper, såsom latens och genomströmning, genom kvantitativ benchmarking. I våra benchmarking-scenarier uppnådde FSF konsekvent högst genomströmning och lägst latens i samtliga testade experiment, medan ADF med standardkonfiguration visade måttlig prestanda på grund av dess I/O-bundna tillståndshanteringsmönster. Våra benchmarkingresultat tillhandahåller kvantitativa mått på prestandaegenskaper och arkitektoniska attribut hos tillståndsbaserade serverlösa tillvägagångssätt, vilket bidrar till förståelsen av dessa system. Resultaten och bidragen från denna avhandling är värdefulla för systemarkitekter, utvecklare och forskare

Nyckelord

Serverlös databehandling, Tillståndsbaserad serverlöshet, Azure Durable Functions, Apache Flink Statefun, Funktion-som-en-tjänst, Tillståndshantering

Acknowledgments

I want to thank my supervisor, Matti Siekkinen, for his patient support, and constructive feedback throughout this process. With his support, I always felt guided and on track. I would also like to extend my appreciation to my supervisor, Ki Won Sung, and my examiner, Slimane Ben Slimane, from KTH Royal Institute of Technology, for their guidance.

I also want to thank my girlfriend for her understanding and emotional support throughout the writing of this thesis.

Finally, I am grateful to my family and friends for their unwavering support throughout this journey.

Stockholm, July 2025
Furkan Ün

Contents

Abbreviations and Acronyms xv

1 Introduction 1
1.1 Background . . . 1
1.2 Problem Statement . . . 2
1.3 Purpose and Goal . . . 3
1.4 Methodology . . . 3
1.5 Delimitations . . . 4
1.6 Structure of the thesis . . . 4

2 Background 7
2.1 Serverless Cloud Computing . . . 7
2.2 Function as a Service: FaaS . . . 8
2.2.1 The Stateless Nature of FaaS and Challenges . . . 9
2.3 Towards Stateful Workloads . . . 10
2.4 Stateful Serverless . . . 13
2.4.1 Orchestrator-Based Stateful Serverless Systems . . . 13
2.4.2 Actor-like Stateful Serverless . . . 15
2.5 Apache Flink Statefun . . . 18
2.5.1 Virtual Addressing and Actor-Like Behavior . . . 19
2.5.2 Fault Tolerance and Idempotency . . . 19
2.5.3 Deployment Flexibility and Rolling Updates . . . 21
2.6 Azure Durable Functions . . . 21
2.6.1 Execution Backends . . . 22
2.6.2 Event Sourcing and Replay . . . 22

3 Performance Evaluation 25
3.1 Scenario: Stateful Checkout Workflow . . . 25
3.2 Experimental Setup . . . 27
3.2.1 Scenario Naming . . . 27

3.2.2 Platform Environments 27
3.3 Azure Durable Functions Setup 27
3.4 Apache Flink StateFun Setup 29
3.4.1 FSF-AzureFunc: Handlers on Azure Functions 31
3.4.2 FSF-K8S: Handlers Within the Kubernetes Cluster . . 32
3.4.3 Flink Statefun Cluster Deployment 32
3.5 Load Generation 34
3.6 Metrics and Data Collection 35
3.6.1 Collected Metrics 36
3.6.2 Apache Flink StateFun Logging 37
3.6.3 Azure Durable Functions Logging 37
3.7 Log Collection and Analysis 37

4 Results and Analysis 39
4.1 Benchmarking Scenarios 39
4.2 RQ1: Architectural and Functional Differences 40
4.3 RQ2: Performance Comparison Under Identical Workload . . 41
4.3.1 Throughput Results 41
4.3.2 EP1 Single Instance Comparison 41
4.3.3 EP3 Single Instance Comparison 42
4.3.4 Latency Performance (EP1) 43
4.3.5 Latency Performance (EP3) 44
4.3.6 Comparative Analysis: EP1 vs. EP3 Performance . . . 46
4.3.7 Overall Trend of Latency Reduction 47
4.3.8 Apache Flink StateFun (K8s) 47
4.3.9 Azure Durable Functions (ADF-Netherite) 48

5 Discussion 49
5.1 Research results 49
5.2 Suggestion for Developers 50
5.3 Limitations 50
5.4 Future Work 51
5.5 Reflection 51

6 Conclusions 53

References 53

List of Figures

2.1 Race condition example caused by two concurrently triggered functions (FunctionA and FunctionB) 12
2.2 Orchestrator based workflow Azure Durable Functions combined with actor-like model Durable Entities, Source: adapted from [10] 14
2.3 Simplified overview of Apache Flink Statefun Architecture . . 18
2.4 Azure Durable Function Orchestrator History Table: Each action is saved in the history table. 22

3.1 Sequence diagram of the checkout workflow 26
3.2 Workload generation and event consumption. 35
3.3 Example Application Insights query 38

4.1 Average Throughput on EP1 (Single Instance) 42
4.2 Average Throughput on EP3 (Single Instance) 43
4.3 Latency distribution comparison for EP1 and EP3 setups. . . . 45
4.4 Latency trend on EP3 - 1000 Events 46
4.5 Mean Latency Trend of All Platforms in EP1 (1 CPU) and EP3 (4 CPU) setups. Solid lines represent EP1 performance, and dashed lines represent EP3 performance. 47

List of Tables

2.1 Stataless FaaS Limitations and Stateful Serverless Motivations 17
2.2 FaaS Challenges and Apache Flink Statefun methodology . . . 20
2.3 FaaS Challenges and Azure Durable Function methodology . . 23

3.1 Namings used for experimental configurations 27
3.2 Summary of Azure Durable Functions Experimental Configurations 29
3.3 AKS Kubernetes node CPU and RAM specifications. 32

4.1 Architectural Comparison of Azure Durable Functions (ADF) and Apache Flink StateFun (FSF) 41
4.2 Latency (ms) Summary for EP1 Single-Node Experiments (Mean, P90, P95, P99) 44
4.3 Latency (ms) Summary for EP3 Single-Node Experiments (Mean, P90, P95, P99) 46

Listings

2.1 Sample Azure Function Configuration with HTTP Trigger . . 8
2.2 ADF parallel function execution implemented in C# 15
2.3 Simplified StockEntity implemented in C#, Azure Durable Entity 16
2.4 Function Addressing in Apache Flink StateFun 19
3.1 application-module.yaml for FSF-AzureFunc scenario 30
3.2 application-module.yaml for FSF-K8s scenario 31
3.3 Kubernetes Deployment YAML for FSF-K8S Handlers 33
3.4 Simplified Kafka workload generator scriptt 34

Abbreviations and Acronyms

ADF	Azure Durable Functions
API	Application Programming Interface
BaaS	Backend-as-a-Service
EP1	Elastic Premium Plan 1
EP3	Elastic Premium Plan 3
FaaS	Function-as-a-Service
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
REST	Representational State Transfer
SSS	Stateful Serverless System

Chapter 1

Introduction

This chapter describes the research question, sets the context of the problem, the goals of this thesis project, and finally outlines the structure of the thesis.

1.1 Background

Today, cloud computing is a fundamental part of the modern software development process and has become a standard approach to deploy and develop new software systems. Companies are increasingly migrating to cloud platforms to benefit from on-demand computing resources with almost zero up-front investment in data centers and physical hardware resources [1, 2].

Early adoption of cloud computing was primarily focused on virtualization, where multiple cloud clients could run their piece of software on virtual machines that shared the same physical hardware [2]. This model relieved companies from on-premise physical hardware management; however, the management and configuration of these virtual resources were still on the cloud users and required developers or system administrators to handle administrative tasks such as applying system upgrades and security patches, monitoring, and scaling machines in response to increasing requests and load [3]. These operational efforts resulted in the development of a new cloud paradigm called serverless cloud computing.

Serverless cloud computing, also mostly known as a Function-as-a-Service (FaaS), was first introduced by AWS with Lamdbda [4] service in 2014. Today, almost all major cloud providers offer similar FaaS services, such as Google Cloud Functions and Azure Functions [5, 6]. In this model, cloud users write a piece of code, or, namely, functions, and cloud providers take the responsibility of the underlying operational aspects, such as execution,

auto-scaling, and monitoring of these functions, instead of the cloud users themselves.

Functions are designed to be stateless and fine-grained. This allows the cloud provider to dynamically provision and scale the functions if the load increases, and the cloud users are charged only for actual execution time rather than pre-allocated compute resources. Besides its cost and operational benefits, serverless functions present certain difficulties, such as a lack of direct function-to-function communication, short execution time limits, and transient failures, which are some of the discussed limitations in the literature [7, 3, 8].

Functions are stateless, which means they are executed in short-lived containers, and once the function's execution is done, the serverless platform de-allocates the resources and has no knowledge about past executions. This allows functions to be highly scalable and works nicely with simple scenarios that take an input and produce an output. However, real-world scenarios usually require the coordination of multiple functions, where multiple small functions come together and build bigger applications. Since functions are short-lived and ephemeral, application state, such as execution progress, partial results, and shared data, needs to be externalized to a storage service or a queue system. However, introducing state and externalizing it to storage services comes with a new set of problems, such as managing shared state access, dealing with retries and duplicate executions, failures, and incomplete executions [9]. Dealing with these problems requires a good understanding of distributed system design, which is against the initial benefits of serverless functions: simplicity and developer productivity.

1.2 Problem Statement

The challenges of adding state to serverless functions led to the initial concept of FaaS evolving into new architectural patterns and advanced systems that we refer to as Stateful Serverless System (SSS) in this thesis.

SSS provides abstractions that hide the challenges of managing and persisting state that we mentioned above while keeping the benefits of serverless functions, such as auto-scaling and usage-based billing. There are various stateful serverless systems that have been proposed in both industry and academia, such as Azure Durable Functions [10], Cloudstate [11], Cloudburst [12], and Apache Flink Statefun [13].

Proposed SSS provide built-in state persistence and management, but they differ in architecture, programming models, state persistence mechanisms, and

consequently, the performance. A systematic comparison of these platforms is lacking. This thesis investigates the following research questions:

- **RQ1: Architectural Qualities:** What are the key architectural differences (e.g., state persistence mechanisms, programming models, and fault tolerance strategies) among stateful serverless systems? How do these differences impact their design and performance?

- **RQ2: Performance Analysis:** Under an identical workload, how do chosen SSS compare in terms of performance metrics such as latency and throughput?

1.3 Purpose and Goal

The purpose of this master's thesis is to investigate and systematically compare SSS that have been proposed to expand the capabilities of FaaS. There is a lack of comparison and evaluation of these proposed SSS side-by-side. This thesis aims to fill that gap by conducting a comparative study of stateful serverless platforms, examining how they provide state persistence and other distributed system features such as fault tolerance, consistency, and execution guarantees. Finally, help developers make informed decisions while choosing between stateful serverless platforms that match their system requirements.

1.4 Methodology

To enable a focused and in-depth investigation, this thesis conducts a comparative analysis on two distinct stateful serverless platforms: Azure Durable Functions (ADF) and Apache Flink StateFun (FSF). This selection is based on their difference in architectural approach and matureness of the platforms. While ADF follows an orchestrator-based methodology and is widely used in production on the Azure Platform [14], FSF represents an actor-like model that can be deployed as a Kubernetes cluster.

To answer RQ1 (Architectural Characteristics), we will conduct a detailed qualitative analysis, which will involve the steps

- **Literature review:** Through evaluation of each platform's official documentation, and review of the relevant research and academic papers related to the chosen SSS.

- **Side by side comparison:** Provide a comparison table for each platform's architectural features and built-in primitives, such as their state persistence mechanisms, fault-tolerance mechanisms (e.g., checkpointing), programming models (e.g., orchestrators, actor-like models), fault tolerance guarantees, and consistency models.

To answer RQ2 (Performance Analysis), we will use a simple stateful checkout scenario implementation to benchmark chosen SSS with varying configurations and loads. Benchmarking is a widely recognized and established methodology to compare different software systems under controlled and reproducible conditions [15].

Section 3.1 describes the checkout scenario and used experimental setups in more detail. For the assessment of RQ2, we will use latency and throughput metrics of the systems under an identical load and compare the obtained results from the experiments.

1.5 Delimitations

While there are various SSSs proposed both in industry and academia, we focused on comparative analysis and benchmarking of two specific SSS: Azure Durable Functions (ADF) and Apache Flink Statefun (FSF) to ensure an achievable research goal within the given time limit of this master's thesis. While other performance metrics, such as cold start times and resource consumption (CPU, Memory), are highly relevant for serverless functions, they are out of the scope of this thesis. We restricted the performance evaluation to latency and throughput.

Moreover, this thesis does not aim to provide a comprehensive benchmarking methodology for all possible use cases but rather focuses on a sample scenario that highlights architectural and performance differences on the chosen SSS.

1.6 Structure of the thesis

This thesis is structured as follows:

– Chapter 2 provides the relevant background information for this thesis by explaining serverless cloud computing, limitations, and transition to stateful serverless systems.

– Chapter 3 focuses on the methodology. In this chapter, we explain the implemented benchmarking scenario and the experimental configurations, such as used tools, versions, load generation, and metric collection.

– Chapter 4 presents the findings for research questions and discusses the obtained results from benchmarking.

– Chapter 5 discusses the results, explains the limitations of the thesis, and future directions.

– Chapter 6 concludes and summarizes the main findings of the thesis.

Chapter 2

Background

In this chapter, we explain serverless cloud computing and discuss its benefits, stateless nature, and limitations. Then, we explain the architectural evolution from stateless FaaS platforms toward stateful serverless systems.

2.1 Serverless Cloud Computing

Serverless cloud computing is a model that enables developers to deploy and run applications without dealing with their operational aspects. According to Datadog's 2023 ”State of Serverless” report [16], the majority of organizations using AWS or Google Cloud had at least one serverless deployment.

Serverless cloud can be used as a comprehensive term for different services [17]. Today, almost all of the major cloud providers offer a set of solutions [18] under the serverless deployment model.

The term ”serverless” does not mean there are no servers involved. There are still servers however, the management of infrastructure is outsourced to the cloud vendors and offered by them as a service. While serverless is often used as an umbrella term for various cloud services, it is most commonly associated with two core models:

- **Function-as-a-Service (FaaS)** allows developers to deploy and execute a piece of application logic, known as functions, in response to an event such as an HTTP call, an event pushed in a queue. In this model, the cloud providers take the responsibility of the entire execution environment and allocate resources for the functions only when they are active and running.

- **Backend-as-a-Service (BaaS)** provides managed services for common

functionalities such as notification services, authentication services, and object storage solutions. In this thesis, we will mainly focus on the FaaS model.

2.2 Function as a Service: FaaS

Function-as-a-Service (FaaS) is one of the most well-known service models of serverless computing. In FaaS, application logic is broken down into small, multiple stateless functions. These functions look similar to regular functions a developer would define in their application logic. However, one difference is that instead of regular method calls, these functions are called as a response to an event, and listen for triggers such as HTTP or a queue trigger. Developers define the triggering mechanism and associate it with the function before deploying it to the cloud environment. After these functions are deployed to a FaaS platform, the cloud provider handles their execution, monitoring, provisioning, and auto-scaling. [3].

Listing 2.1 Sample Azure Function Configuration with HTTP Trigger

```
1  "bindings": [
2  {
3  "authLevel": "anonymous",
4  "type": "httpTrigger",
5  "direction": "in",
6  "name": "req",
7  "methods": ["post"]
8  },
9  {
10 "type": "http",
11 "direction": "out",
12 "name": "$return"
13 }
14 ]
```

Trigger can be defined in different ways. Listing 2.1 shows a function configuration in JSON format triggered by an incoming HTTP POST request and returns an HTTP response. There are various usage scenarios for FaaS, such as web and REST API serving, video processing [19], and machine learning [20].

Serverless model, particularly FaaS, brings several benefits to both cloud vendors and application developers.

- **Focus on the application code:** From a developer's point of view, since the management of resources is handled by the cloud providers, FaaS simplifies code delivery and allows them to focus more on the application logic.
- **Dynamic Resource Allocation:** Functions are small, light-weight, and event-driven. These characteristics of functions allow better load balancing and resource utilization compared to traditional virtual machine deployments [3]. Compute resources like CPU and memory are automatically allocated when a function is triggered and deallocated afterward. As a result, FaaS prevents over-provisioning of resources [8]. Also, scaling happens automatically based on demand. This includes both horizontal scaling (running multiple instances in parallel) and scaling down to zero when there is no execution.
- **Usage-based billing:** One of the key characteristics of FaaS is that billing is based on actual usage, which means cloud users are only billed when functions are executed or triggered. Billing can differ between cloud providers; however, typically it is based on the number of invocations, duration of function execution, and memory consumption [21, 22]. Thus, choosing FaaS can be cost-efficient for applications, especially for those with variable traffic.

2.2.1 The Stateless Nature of FaaS and Challenges

While the serverless functions provide various benefits, they also come with their inherent challenges [7, 3, 8]. Functions are designed to be stateless, which means they are executed in isolation and independently. [7]. This stateless nature of functions enables them to be highly scalable since any function instance can process the incoming requests. However, it can introduce several limitations, especially for the workloads that require a persistent state or state sharing.

- **No built-in state:** Functions are executed in short-lived ephemeral containers. Once the function execution is done, the serverless platform shuts down the container, which in turn causes any in-memory state to be lost if it is not persisted in an external database or a storage system.[23].

- **Execution time limit:** Usually, serverless platforms put a maximum execution time for the functions [24], while it is a maximum of 15 minutes for AWS [25], single Azure Functions time out after 5 minutes (configurable for certain plans) [26]. This bounded execution time can limit their usage, especially in long-running or complex multi-step workflow scenarios, such as data processing pipelines. Intermediate results need to be serialized to an external system such as file storage(S3) or key-value databases (DynamoDB).

- **Internal Communication:** Executed functions are usually put behind a NAT. They can make outbound calls; however, it is not possible to address them for the point-to-point network communication [23]. Thus, function-to-function communication needs to happen through an external storage or a queue, which is slower compared to point-to-point communication.

- **External Storage Latency:** Since the state persistence or function communication happens through external services such as key value storage, functions frequently need to read and write back to state changes. These frequent read and write operations introduce additional latency due to I/O overhead. Moreover, using cloud storage for communication is not only a problem for performance, but it can be costly. As highlighted in [23], using DynamoDB for internal communication of 1000 lambda functions costs approximately $450 per hour.

2.3 Towards Stateful Workloads

The challenges mentioned above are addressed mainly by decoupling the state from the computation and adopting the FaaS towards an architecture that externalizes state to the persistent storage services[24]. However, calling external services and managing state come with a new set of problems that affect application design and maintainability. First of all, these storage services are managed separately, and the characteristics of these storage services need to be considered carefully, such as efficiency, consistency, scalability, and fault tolerance.

- **Incomplete & Partial Execution** Serverless functions run in a distributed environment, and failures can occur due to various reasons such as runtime exceptions, function timeouts, and even hardware

issues [9]. If a function stops execution in the middle of a multi-step storage update, it may leave the storage inconsistent. There should be a mechanism to handle partial executions.

- **Execution Guarantees**: Serverless platforms usually offer weak execution guarantees, such as at least once execution for queue-based triggers and event-based triggers. It means two function instances can be triggered for the same event, and if it is not considered while designing the functions, it might cause side effects. For example, in an e-commerce checkout scenario, a customer could be charged twice or receive the order confirmation notification twice.

- **Concurrency/Parallellism** FaaS functions are known for their high scalability. This can cause a race condition issue. For example,in a simple stateful scenario where a function is incrementing a counter value that is stored in a key-value store, if two functions are triggered at the same time, it might cause the counter value to be overwritten by each other. As shown in Figure 2.1:

  1. Both functions read the current counter value (e.g., 10) simultaneously.
  2. Each function independently increments the value from 10 to 11.
  3. Both functions write the new value (11) back to the store.

  In this scenario, we would expect the final value to be 12, but because of concurrent overwrite, it remains 11. To prevent such race conditions, there should be a concurrency control mechanism such as optimistic locks or leases.

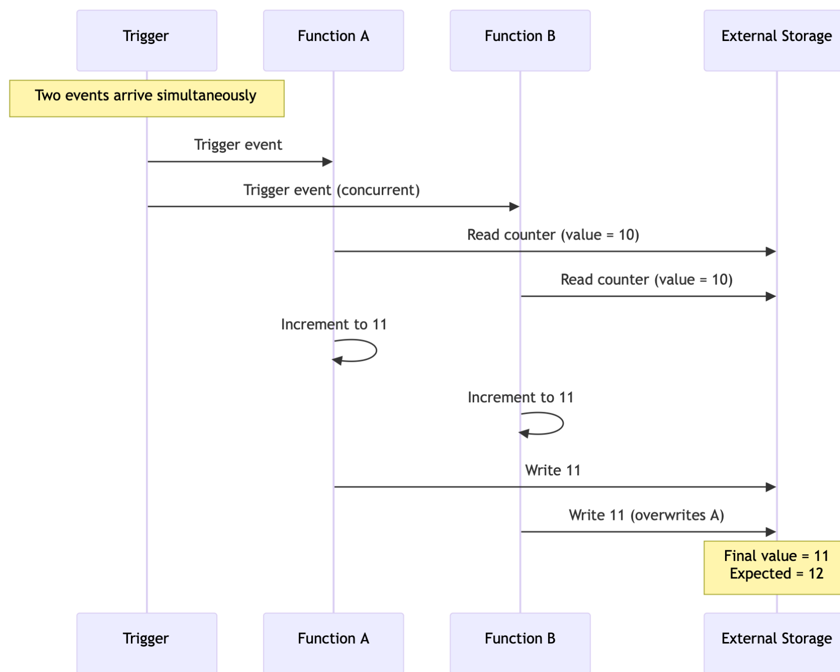


Figure 2.1: Race condition example caused by two concurrently triggered functions (FunctionA and FunctionB)

2.4 Stateful Serverless

As we mentioned in Section 2.3, separating the state and functions needs careful design and deep understanding of the distributed system features such as fault tolerance, concurrency, and consistency [23]. Designing and implementing such functions and systems can be a difficult task for developers, which is against one of the core benefits of the FaaS itself: *simplicity*.

The stateless nature of FaaS and difficulties of adapting serverless to real-world applications and complex workflows led to the evolution of the initial concept of ”Function as a Service” to more advanced platforms and architectures [9], which we refer to as **stateful serverless systems** in this thesis. Two main approaches have been proposed:

1. **Orchestrator Based**, where the execution flow of the functions is managed by a central coordinator.
2. **Actor-Based Programming Models**, where the state is treated as a fundamental part of the execution model. In this model, the state and business logic are bundled into addressable units called actors. Each actor maintains its own private state. Actors can interact with each other asynchronous messsage passing by sending a message to the address of other actors without relying on a central orchestrator.

2.4.1 Orchestrator-Based Stateful Serverless Systems

As we previously discussed in 2.2.1, serverless functions are designed to be short-lived, and they have a maximum execution time limit, which can be a limiting factor for long-running and multi-step workflows. To overcome this and to enable long-running workflows, various workflow orchestrators have been proposed. These workflow orchestrators enable developers to define, orchestrate, and execute complex workflows that require communication of multiple stateless FaaS.

Workflow orchestrators manage the communication between multiple serverless functions through a centralized service called the orchestrator. The orchestrator keeps a holistic view of the whole workflow process, such as running functions and outputs of intermediate function calls by using state machines or persistent checkpointing mechanisms. Critically, these platforms handle the aspects of distributed features such as error recovery and a retry mechanism for failed executions. Also, they provide various

function invocation patterns such as sequential execution, parallel execution (fan-out/fan-in), and even manual human intervention [10].

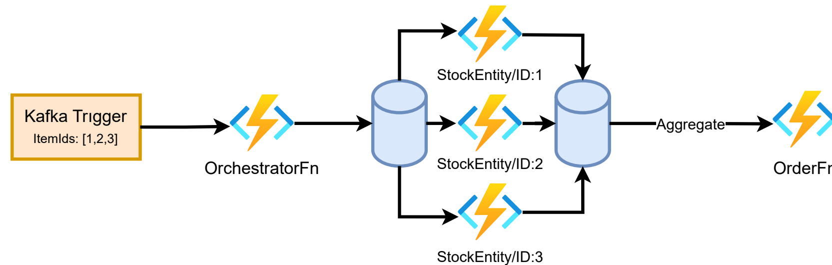


Figure 2.2: Orchestrator based workflow Azure Durable Functions combined with actor-like model Durable Entities, Source: adapted from [10]

Figure 2.2 shows an example of an orchestrator-based stateful serverless workflow that checks the stock amounts for each product with a fan-out/fan-in pattern before creating an order. Similar to serverless functions, orchestrators are event-driven. In this specific example, the orchestrator listens to Kafka events, which contain the input for the workflow. Once the orchestrator starts running, it triggers execution of three function calls asynchronously and in parallel to check the stock amounts of the products by calling StockEntity with their IDs. Then, it combines the responses returned from these calls and saves the output of each call to an external storage. Finally, it calls the OrderFunction with these aggregated results and completes the workflow.

Listing 2.2 provides the simplified implementation of the workflow shown in Figure 2.2. Orchestrator keeps the list of asynchronous calls made to StockEntities and awaits them in a non-blocking way. Once all tasks are completed, the orchestrator continues the execution. Line 5 shows how each StockEntity is addressed by ID.

Today, almost all cloud providers offer a workflow orchestrator as a service:

- **Azure Durable Functions (ADF)** – developed by Microsoft, and workflows can be defined as code in various programming languages such as C #, Python, and Powershell [10, 27]. It is based on the open-source Durable Task Framework [28]. We explain the ADF more detailed in Section 2.6.

- **AWS Step Functions** – Developed by Amazon and uses JSON-based domain-specific language (DSL) to define workflows [29]. It internally uses state machines and supports parallel execution, function chaining, and automatic function retries [27, 30].

Listing 2.2 ADF parallel function execution implemented in C#

```
1  var retractTasks = new List<Task<bool>>();
2
3  foreach (var item in request.Items)
4  {
5      var stockEntity = new
        ↪  EntityInstanceId("StockEntity",
        ↪  item.ItemId);
6      retractTasks.Add(
7          context.Entities.CallEntityAsync<bool>(
8              stockEntity, "Retract",
9              new RetractRequest { Amount =
                ↪  item.Quantity }));
10 }
11
12 bool[] results = await Task.WhenAll(retractTasks);
```

2.4.2 Actor-like Stateful Serverless

Actor-like stateful serverless systems treat the state as a built-in concept. In this approach, state and behavior are encapsulated in addressable units called an Entity.

Entities are virtual, which means they are not created, but the system behaves as if the entity instances had always existed. They are callable through virtual addresses. The virtual address for each entity can be derived by typename/id. For example, a Product entity with ID `1` can be addressed with `Product/1`.

Each entity is responsible for its own local and private state, and the state is not shared between different entities. The platform itself handles entity lifecycle transparently, entities are loaded into memory (deserialized) when they are invoked, and deallocated from the memory (serialized back to persistent storage) when they are idle [31].

Listing 2.3 shows a StockEntity example where the StockEntity keeps the state (stock amount) for a specific product and provides the behavior (AddStock, RetractStock) to modify the state.

The following systems follow the actor-like models:

- **Apache Flink Stateful Functions** uses Apache Flink stream platform and combines stream processing with the actor-like model. [32]

Listing 2.3 Simplified StockEntity implemented in C#, Azure Durable Entity

```
    public class StockEntity : TaskEntity<int>
{
    protected override int
      ↪ InitializeState(TaskEntityOperation op){
        return 100
    };

    public bool RetractStock(int amount)
    {
        State -= amount;
    }

    public void AddStock(int amount)
    {
        State += amount;
    }

    public int Get() => State;
}
```

- **Azure Durable Entities:** Added in V2 of the ADF. Durable Entities follows the actor-like programming model, which is adopted from the Orleans virtual actor model [33, 34].

Stateful serverless systems provide higher-level abstractions that hide the complexity of state management [9]. Thus, they simplify the development of stateful serverless applications without losing the original promise of serverless functions, such as auto-scaling and load-based billing. Table 2.1 provides a summary of the stateless FaaS limitations and how SSS addresses them:

Table 2.1: Stataless FaaS Limitations and Stateful Serverless Motivations

Stateless FaaS Challenge	**Consequence**	**Stateful Serverless Methodology**
Ephemeral, short-living containers	Application state is lost between function calls if it is not stored in external storage.	SSS comes with built-in state management mechanisms that allow function execution in a reliable and fault-tolerant way. If a failure occurs during execution, application state can be rebuilt by using the checkpointing mechanism [35] or re-executing the history of events (replay-based [9] model).
Externalize state to a persistent storage service	Frequent access to state (reading and writing back to external storage) can introduce I/O overhead to the system. This might cause a performance bottleneck and additional storage costs, especially in cloud offered storage services. Another consequence of externalizing state is that if the state is shared between multiple functions (Section 2.3), concurrent access to shared state needs to be handled at the application level or at the database level.	Provides optimizations for the state access with various methodologies such as co-location of state and compute (actor-like) [32], built-in caching mechanism, and group committing in Netherite [14]. SSS also offers strong execution guarantees such as exactly-once processing and data consistency guarantees with locking mechanisms [14] and sequential access to local state (actor-like models).
Function to Function Communication	Functions that have a data dependency communicate with each other through an external system. For example, if a function A needs a result of function B as an input, then function A needs to write the result to a file, a database, or a queue system [7].	Provides built-in messaging (virtual-addressing in actor-like models) and function execution patterns such as function-chaining, fan-out/fan-in pattern [10] which allows easy communication between functions, especially in complex workflows (orchestrator-based models).

2.5 Apache Flink Statefun

Apache Flink Stateful Functions (StateFun) is a framework built on top of a streaming platform. While Apache Flink itself is a stream processing engine, StateFun extends its capabilities and combines the strengths of stream processing (low-latency processing, strong consistency guarantees) with the benefits of serverless functions, such as auto-scaling, zero-scaling, and rolling updates [36]. The StateFun architecture has the following four key components:

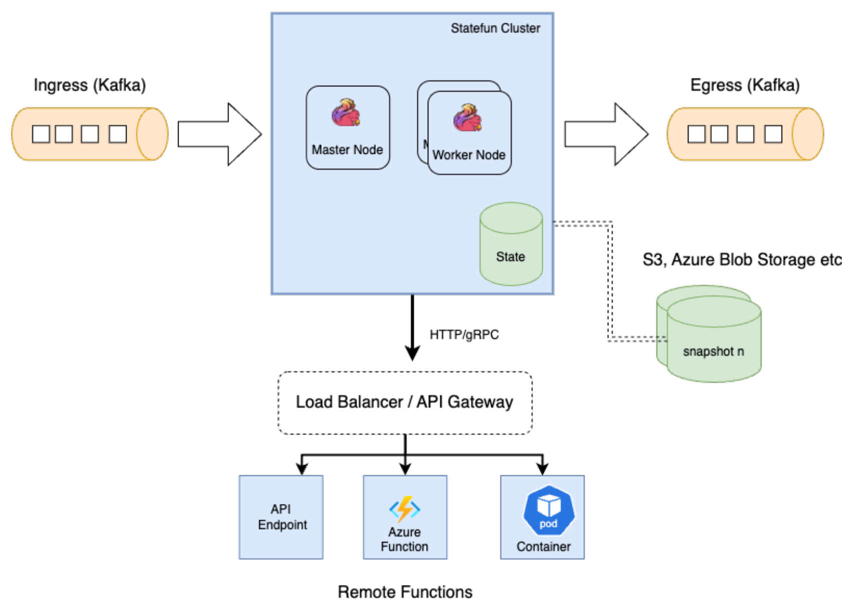


Figure 2.3: Simplified overview of Apache Flink Statefun Architecture

- **Ingress:** is the system's entry point where the Flink cluster listens for events from external systems such as a Kafka topic or an HTTP call to trigger functions.
- **StateFun Cluster:** Acts as the event-driven database for the system and is responsible for the state management, calling remote functions, and checkpointing for fault tolerance.
- **Remote Function Handlers:** Stateless FaaS or a microservice that implements the actual application logic.

- **Egress:** responsible for sending messages and intermediate results to the external systems, such as making a GRPC call or writing results to a Kafka topic, by using Flink Connectors.

2.5.1 Virtual Addressing and Actor-Like Behavior

Statefun follows the actor-like model. Developers write functions that contain the state and the behavior, and these functions can communicate with each other asynchronously without requiring any central orchestrator. Functions are virtually addressed by a **(FunctionType, ID)** :

- **FunctionType:** The type or class of the function (e.g., `StockFunction`).
- **ID:** A unique identifier within that type can be thought of as a primary key in a relational database table (e.g., `"PRODUCT-ID-1"`).

Listing 2.4 Function Addressing in Apache Flink StateFun

```java

RequestItemMessage requestItem = new
    RequestItemMessage(quantity);

Message stockRequest = MessageBuilder
        .forAddress(StockFn.TYPE, productId)
        .withCustomType(requestItem)
        .build();

context.send(message);

```

Messages are routed based on this virtual address. Listing 2.4 shows an example of how functions call each other in FSF. In Line 4, the function first prepares the messages that need to be sent to StockFn by building the message object with its virtual address (Line 5) and the payload (Line 6). Finally, in Line 9, it uses the context method to send this message to StockFn.

2.5.2 Fault Tolerance and Idempotency

Statefun cluster itself acts as an event-driven database between the functions. All the communication, such as messages, comes from ingress, state changes,

and function calls, goes through the Statefun cluster [32]. Statefun provides fault tolerance using Flink's checkpointing mechanism. The cluster takes a snapshot of the system at regular intervals and asynchronously saves it to a persistent storage system, such as S3 or Azure Blob Storage. The frequency of the interval can be set by developers while configuring the cluster. In case of failure, the snapshot contains the consistent view of the whole system, which can be used to restore the system from the last available checkpoint. [32].

Statefun uses the stateful stream processing approach where state is co-located with the functions [32]. StateFun worker nodes listen to incoming events from the ingress service and route messages to the target function based on the message key. The function invocation contains the ingress message and the state read from the local state. Any state mutations made during the function call are sent back to the worker nodes as a response. Then, the local state is updated with all the mutations received in the response. Since each function call is made with the context information, such as the current state of the entity and the ingress message, function calls are idempotent by design. This design ensures that replays or retries (after failures) do not cause any side effects. Thus, developers do not need to pollute their application logic to deal with consistency problems and additional checks, as these are handled by the system itself.

Table 2.2: FaaS Challenges and Apache Flink Statefun methodology

Challenge in Stateless FaaS	**Apache Flink StateFun Methodology**
Incomplete & Partial Execution	StateFun uses Flink's state consistency and checkpointing mechanisms. If a function fails unexpectedly during the execution, the application state can be recovered from the checkpoints.
Execution Guarantees	StateFun guarantees exactly-once message processing. Also, remote function calls are idempotent by design, as a result, there are no side effects from retries, and failures
Concurrency and Race Conditions	Each function instance has its own private and local state, which guarantees atomic updates and eliminates race conditions between the functions.
Externalized State Complexity	State is passed along with the function call and automatically persisted by the framework.

2.5.3 Deployment Flexibility and Rolling Updates

Another key architectural feature of StateFun is the physical separation between the function logic and the state management layer. Function invocation contains all the necessary context and state as input, which means functions remain stateless from the cluster's point of view [37]. This decoupling enables Statefun to work perfectly with FaaS and allows horizontal scaling and rolling updates without actually affecting the cluster itself. [32]

2.6 Azure Durable Functions

Azure Durable Functions (ADF) is an extension of Azure Functions that enables defining long-running, stateful serverless functions. ADF provides a model where developers define orchestrator functions and workflows as code with various programming languages [10], while the platform handles state persistence, retries, and fault tolerance transparently. ADF allows to define three core function types [9]:

- **Orchestrator Functions:** Define the steps of a workflow as a code. Orchestrator functions can call activity functions, wait for external events(human approval), and orchestrate parallel execution such as fan-out/fan-in [10]
- **Activity Functions:** Stateless Azure Functions that can be executed by the orchestrator.
- **Entity Functions:** Azure Durable Entity functions adopt the Orleans virtual actor model [33] where each entity has its own persistent state and virtual address. Orchestrators can interact with entity functions in two ways:
  - **Call an entity** (waits for a response),
  - **Signal an entity** (fire-and-forget).

  All invocations made from the same orchestrator to an entity are queued and executed sequentially, which prevents the race condition issues. Listing 2.3 demonstrates a simplified implementation of a StockEntity, which provides operations to add, retract, and get stock.

2.6.1 Execution Backends

Azure Durable Functions can be configured with different storage providers with a simple change in the configuration file [38]. Each of these options uses a different storage mechanism, which in turn affects the performance.

- **Default Table Storage Backend:** The default backend is the default configuration if it is not explicitly set to another storage provider by developers. It uses Azure Storage Tables and Azure Queues to store orchestration events and entity states.

- **Netherite Backend:** Netherite is an optimized backend option that uses Azure Event Hubs and a log-based state persistence. It keeps the orchestration state in memory and asynchronously persists it as logs to a blob storage. Netherite provides more optimized storage access by using FASTER key-value storage and reduces storage access by using a method called group commit, which allows multiple orchestration events to be saved as a group within a single storage access [14].

IsPlayed	_Timestamp	EventType	ExecutionId
false	2025-06-14T19:22:56.1615970Z	OrchestratorStarted	18be7a4cd62a4893a9f028f37
false	2025-06-14T19:22:55.0496280Z	ExecutionStarted	18be7a4cd62a4893a9f028f37
false	2025-06-14T19:22:57.2563120Z	EventSent	18be7a4cd62a4893a9f028f37
false	2025-06-14T19:22:57.2563370Z	EventSent	18be7a4cd62a4893a9f028f37
false	2025-06-14T19:22:57.2563640Z	OrchestratorCompleted	18be7a4cd62a4893a9f028f37
false	2025-06-14T19:23:02.8930680Z	OrchestratorStarted	18be7a4cd62a4893a9f028f37
false	2025-06-14T19:23:02.2461840Z	EventRaised	18be7a4cd62a4893a9f028f37
false	2025-06-14T19:23:05.5270540Z	OrchestratorCompleted	18be7a4cd62a4893a9f028f37

Showing 1 to 17 of 17 cached items   1   Load more

Figure 2.4: Azure Durable Function Orchestrator History Table: Each action is saved in the history table.

2.6.2 Event Sourcing and Replay

ADF follows the event sourcing pattern by saving every action the orchestration takes (e.g., OrchestrationStarted, EventRaised, OrchestratorCompleted) as an event in the History table. (Figure 2.4). This append-only history allows reliable execution and fault tolerance. In case of failure, the ADF runtime re-executes the history table, if the task is already completed (activity function called), instead of re-executing it uses the saved output from the history table. The replay mechanism allows the orchestrator to rebuild its local state and continue from where it left off. [9]

Table 2.3: FaaS Challenges and Azure Durable Function methodology

Challenge in Stateless FaaS	**Azure Durable Functions Methodology**
Incomplete & Partial Execution	ADF saves the progress and events that each orchestrator takes to a history table. Orchestrators can resume where they left off, which prevents partial workflows.
Execution Guarantees	Replay mechanism guarantees exactly-once processing for the calls that are wrapped into an activity function
Concurrency and Race Conditions	Durable Entities provide actor-like isolation where each entity processes messages one at a time and supports locking in C#.
Externalized State Complexity	Application state is automatically persisted by the ADF runtime.

Chapter 3

Performance Evaluation

This section describes the methodological approach we used to evaluate and compare the Azure Durable Functions and Apache Flink Statefun with a varying workloads in a simplified e-commerce checkout scenario. We mainly focus on empirical performance, where we measure the end-to-end latency and throughput of the systems for each experimental scenario.

3.1 Scenario: Stateful Checkout Workflow

In experiments as a benchmarking scenario, we used a stateful checkout workflow that implements business logic commonly found in real-world e-commerce applications. We adopted the scenario from the FSF official documentation [39] with minor adjustments and implemented it in both systems. The scenario has these steps: first, the client creates a checkout request with items they want to buy, then the system verifies the stocks of each requested item before creating an order. Finally, if stock verification is successful, the receipt is generated as a summary of the checkout request. Figure 3.1 shows the detailed sequence of this workflow.

We chose the checkout workflow not only for its real-world relevance but also it suites to investigate certain characteristics of stateful serverless platforms that we benchmark. Firstly, the scenario itself requires:

- *State persistence* — such as order status, and per-item stock amount, which perfectly matches with an actor-like model where each actor, in this scenario, product, is addressed by a virtual id and its own private state (stock amount).

Secondly, the scenario needed to complete multiple steps before a successful

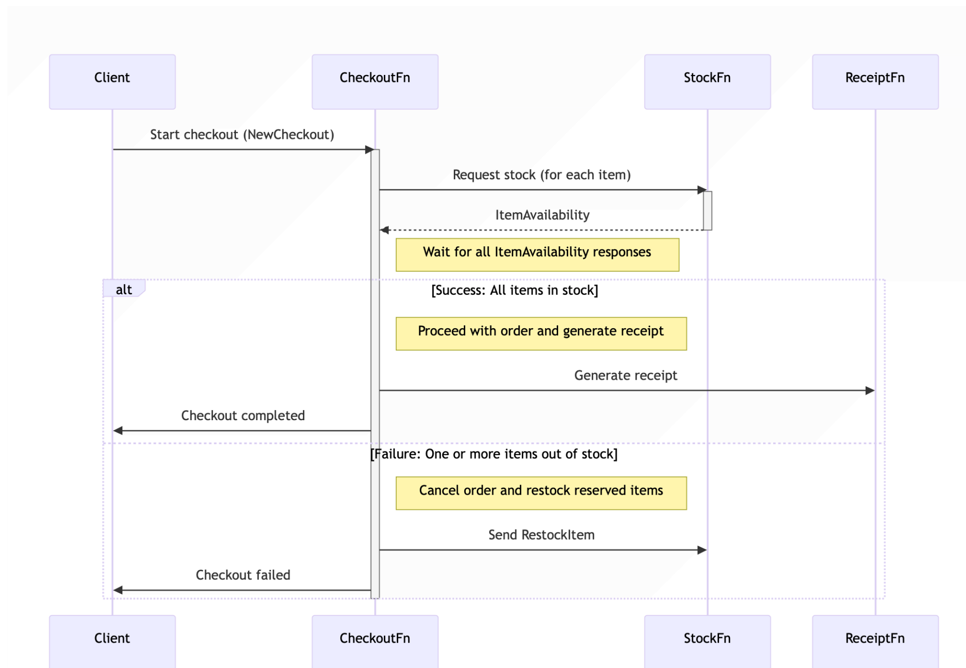


Figure 3.1: Sequence diagram of the checkout workflow

checkout, such as a stock availability check, receipt generation, which allows us to use function execution patterns provided by the platforms, such as fan-out/fan-in[10] pattern, which we used to check stock levels for each item in parallel.

Moreover, this scenario allows us to collect the metrics in a simple and understandable way:

- *End-to-end latency* — we measure it from checkout initiation to receipt generation
- *Varying workload generation* — allows us to benchmark systems with increasing checkout requests and observe performance under increasing load.

This workflow is well-suited to evaluate the underlying runtime features of platforms, state access patterns, and orchestration logic, which makes this workflow an effective benchmark scenario for comparing the performance of chosen stateful serverless platforms.

3.2 Experimental Setup

This subsection gives an overview of experimental setup such as used tools and cloud plans, deployment configurations, and finally the naming for different experimental setup used in the evaluation of Azure Durable Functions and Apache Flink StateFun.

3.2.1 Scenario Naming

Table 3.1 shows the namings that we refer to each experimental configuration the rest of this thesis.

Scenario Name	Description
ADF-Default	Azure Durable Functions with default Table Storage backend
ADF-Netherite	Azure Durable Functions with Netherite backend
FSF-K8S	Flink StateFun with remote handler deployed in AKS cluster
FSF-AzureFunc	Flink StateFun with remote handler deployed as Azure Function

Table 3.1: Namings used for experimental configurations

3.2.2 Platform Environments

To achieve a fair and reproducible comparison, we deployed each platform separately to the Azure cloud environment. During all of the experiments, we used Microsoft Azure Cloud as a cloud provider, specifically in the North Europe region.

3.3 Azure Durable Functions Setup

Azure Durable Functions supports various programming languages [10]. We implemented the checkout benchmark workflow using the .NET Isolated Process runtime with C#, as we could find practical examples implemented with C# in the official documentation. ADF Netherite backend is only supported in Premium and Dedicated hosting plans on Azure, this directly affected our choice of machine specifications.

- **Provider:** Microsoft Azure
- **Region:** North Europe

- **Runtime:** Azure Functions v4
- **Programming Language & SDK Version:**(Isolated process, .NET 8)
- **Hosting Plan:** Elastic Premium EP1
- **Instance Specification:** 1 vCPU, 3.5 GB RAM per instance

To analyze the effect of increased resources on performance, we used two different premium plans (EP1, EP3) for the ADF function scenarios:

- **Single-Node (EP1 × 1)**: 1 vCPU and 3.5 GB memory, 1 instance
- **Single-Node (EP3 × 1)**: 4 vCPU and 14 GB memory, 1 instance

It was critical that the number of function instances remain fixed during our experiments. To achieve this, In both configurations, we explicitly turned off auto-scaling to prevent cold-start delays and inconsistent results by setting the `maximumInstanceCount` and `alwaysReadyInstanceCount`.

While ADF can be configured with various backend storage providers such as *Azure Storage Table*, *MSSQL*, and *Netherite*. We have focused mainly on two variants:

- **ADF-Default** In this scenario, we used the default storage provider, which is the default configuration if not set by users explicitly. It uses *Azure Table Storage* for managing orchestration state and history. In our experiments, the storage account was configured with the cost-effective SKU *Standard_LRS*.
- **ADF-Netherite:** Azure durable functions uses the Netherite backend storage provider for task orchestration, which uses Azure Event Hubs for internal communication. During our experiments, we started and used a minimum cost-effective configuration *1 Throughput Unit (TU)* for Event Hub.

During the checkout workflow implementation, we used the following tools in local development environment before deploying benchmarking scenarios to the cloud environment.

- **Visual Studio Code (VSCode)** We used *VSCode* as the integrated development environment (IDE) for all our experiments. VSCode provides extensions that make it easier to work with Azure Functions, such as creating project templates and deploying functions to the Azure environment.

- **Azure Functions Core Tools** (v4) for local execution and emulation of the Azure Functions runtime environment.
- **Azure Storage Emulator** (Azurite v3.28 via Docker) for local testing with the default storage provider configuration.
- **Azure Storage Explorer** to query and track the orchestration history and state tables during tests.
- **.NET 8 SDK** for building and running the isolated process functions locally.
- **Postman** for manual API testing and workflow triggering during development.

This local environment setup enabled us to test the benchmark scenario before making any deployment to a cloud environment with almost zero cloud billing cost.

It is important to note that the local environment was only used for development and initial testing purposes before deploying scenarios to the cloud environment. The performance evaluations presented in this thesis were conducted in the cloud-based environment presented in the Table 3.2.

Table 3.2: Summary of Azure Durable Functions Experimental Configurations

Scenario	Plan	Total Resources	Backend Storage	Event Hub TU
ADF-Default (Single-Node)	EP1	1 vCPU, 3.5 GB RAM	Azure Table Storage(Standard_LRS)	–
ADF-Netherite (Single-Node)	EP1	1 vCPU, 3.5 GB RAM	Netherite (via Event Hubs)	1 TU
ADF-Default (Single-Node)	EP3	4 vCPUs, 14 GB RAM	Azure Table Storage(Standard_LRS)	–
ADF-Netherite (Single-Node)	EP3	4 vCPUs, 14 GB RAM	Netherite (via Event Hubs)	1 TU

3.4 Apache Flink StateFun Setup

In Apache Flink StateFun experiments, we implemented the identical checkout benchmark scenario mentioned in Section 3.1 to provide a direct comparison with Azure Durable Functions. The scenario was implemented with the Java SDK, using Apache Flink StateFun version 3.2.0, which is the latest stable release when we were doing the experiments. In Statefun experiments, we used the following tools and versions:

- **Programming Language & SDK:** Java with Apache Flink StateFun SDK version 3.2.0

- **Java Development Kit (JDK):** Version 21 (OpenJDK)
- **Build Tool:** Apache Maven 3.9.4
- **Containerization:** Docker version 24.0.5
- **Orchestration Platform:** Kubernetes version 1.29
- **Local Development:** Minikube version 1.32.0 (used for local testing)
- **Cluster Deployment:** Helm version 3.14.0
- **Logging Framework:** Log4j2 for structured logging

StateFun supports calling user-defined functions as remote function handlers using its `RequestReply` protocol. We used the exact same function implementation on two different deployment scenarios: one where the function handlers were hosted on Azure Functions (FSF-AzureFunc), and another where we deployed the function handlers as pods within the same Kubernetes cluster as Flink (FSF-K8S). The key difference between these two deployments is the network proximity. Since FSF-AzureFunc is deployed to a public cloud, it introduces additional network latency. Also, FSF-K8s deployed function handlers use Undertow HTTP server, which allows async HTTP calls. However, in FSF-AzureFunc deployment, the underlying HTTP server is managed by the Azure platform. In terms of performance, we expect the FSF-K8S deployment to outperform FSF-AzureFunc because of reduced network latency.

Listing 3.1 application-module.yaml for FSF-AzureFunc scenario

```yaml

kind: io.statefun.endpoints.v2/http
spec:
  functions: com.example/*
  urlPathTemplate:
    ↪ https://flinkfunction.azurewebsites.net/api
  /checkout #Azure URL
  transport:
    type: io.statefun.transports.v1/async
```

To enable the Flink cluster to invoke the specified scenarios, we configured the Flink statefun cluster with a scenario-specific *application-module.yaml*

file. This file defines how the statefun cluster discovers and routes the incoming messages to the corresponding remote function handlers. Each *application-module.yaml* is mounted as a ConfigMap for the deployed scenario. In both scenarios, we used the exact same transport protocol (asynchronous HTTP), while only the endpoint URL differed for the specified scenario. For the Azure-deployed function handler scenario, the URL pointed to the HTTP endpoint provided by Azure (Listing 3.1).

In the Kubernetes deployed scenario (FSF-K8S), we exposed the function handlers internally with a Kubernetes `ClusterIP` service, and pointed the URL to this internal Kubernetes service (Listing 3.2).

Listing 3.2 application-module.yaml for FSF-K8s scenario .

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: application-module
5    labels:
6      app: statefun
7  data:
8    module.yaml: |+
9      kind: io.statefun.endpoints.v2/http
10     spec:
11       functions: com.example/*    ##Route all the
          ↪  functions under this namespace to
12       urlPathTemplate:
          ↪  http://shopping-cart-functions:1108
13       /{function.name} ##ClusterIP
14       transport:
15         type: io.statefun.transports.v1/async
```

The deployment environments and platform configurations used in these two different deployment scenarios:

3.4.1 FSF-AzureFunc: Handlers on Azure Functions

In this scenario, we deployed the Java-based StateFun function handlers as HTTP-triggered Azure Functions. Flink StateFun cluster triggers these handlers using an async HTTP call.

- **Runtime:** Azure Functions v4

- **Programming Language & SDK Version:**(Java, 21)
- **Host OS:** Linux
- **Hosting Plan:** Elastic Premium EP1, Elastic Premium EP3

We used identical hosting plans (EP1) and instances of the Azure Durable Functions setup (Section 3.3) for both single scenarios (EP1 and EP3) to ensure a comparable setup. However, this scenario is excluded from the benchmarking results, which is further discussed in the limitations section.

3.4.2 FSF-K8S: Handlers Within the Kubernetes Cluster

In this scenario, first, we packaged the scenario implementation as a Docker image and pushed it to the Azure Container Registry to make it accessible inside the Kubernetes Cluster. Then, the function handlers are deployed as pods within the same Azure Kubernetes Service (AKS) cluster that hosts the Flink JobManager and TaskManagers. Different from the FSF-AzureFunc scenario, each pod runs an embedded Undertow HTTP server that listens to requests coming from the Flink StateFun runtime via the `RequestReply` protocol. Listing 3.3 shows the Kubernetes function handler deployment specification.

We adjusted the function handler deployment's CPU and memory resources to the Azure Functions EP1 specifications for fair comparison.

3.4.3 Flink Statefun Cluster Deployment

In both the FSF-AzureFunc and FSF-Kubernetes scenarios, we deployed an identical Flink cluster on Azure Kubernetes Service (AKS) with a single JobManager pod and two TaskManager pods. Table 3.3 provides the AKS cluster node specifications.

- **JobManager (Master) Configuration:**

Node Pool Name	VM Size	Node Count	vCPU	Memory (RAM)
`systemnodepool`	Standard_DS2_v2	1	2	7 GB
`standardpool`	Standard_DS3_v2	1	4	14 GB

Table 3.3: AKS Kubernetes node CPU and RAM specifications.

Listing 3.3 Kubernetes Deployment YAML for FSF-K8S Handlers

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: checkout-functions
spec:
  replicas: 1
  selector:
    matchLabels:
      app: checkout-functions
  template:
    metadata:
      labels:
        app: checkout-functions
    spec:
      containers:
        - name: checkout-functions
          image:{{.Values.functions.image}}
          ports:
            - containerPort: 1108
          #  to match EP1 specs
           resources:
             limits:
               cpu: "1"
               memory: "3.5Gi"
```

,

- JVM Heap: 1 GB
- Container Memory: 1.5 Gi

- **TaskManager (Worker) Configuration:**
  - JVM Heap: 1 GB
  - Container Memory: 1.5 Gi

3.5 Load Generation

To evaluate the performance of the scenarios mentioned above, we create checkout events `Checkout` and publish them to a shared Kafka topic named `checkout-events` as controlled batches such as 100, 500, and 1000.

Each generated checkout message consists of two randomly selected items. Before we publish the event to the Kafka topic, every message is assigned a unique `requestId`, a timestamp, and a reference to the experiment ID for later analysis and end-to-end latency tracking. Especially, adding the timestamp at event creation was critical for the end-to-end latency tracking. Listing 3.4 shows a simplified version of the workload generator script.

Listing 3.4 Simplified Kafka workload generator scriptt

```
def build_checkout_event():
    selected_items = random.sample(ITEM_POOL, k=2)
    return json.dumps({
        "requestId": str(uuid.uuid4()),
        "experimentId": "EP1_4_NODE_JAVA_100_1",
        "clientTimestamp": int(time.time() * 1000),
        "message": {
            "userId": str(uuid.uuid4()),
            "items": [
                {"itemId": selected_items[0],
                 ↪  "quantity": random.randint(1,
                 ↪  3)},
                {"itemId": selected_items[1],
                 ↪  "quantity": random.randint(1,
                 ↪  3)}
            ]
        }
    })

producer = Producer({ "bootstrap.servers": "..." })

for _ in range(EVENT_SIZE):
    producer.produce(
    topic="newcheckout-events"
    ,value=build_checkout_event()
    )
```

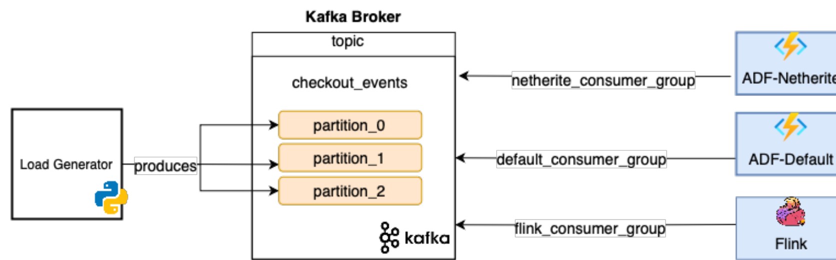


Figure 3.2: Workload generation and event consumption.

While Kafka is the default ingress system for the Apache Flink Statefun, we chose to use Kafka to trigger Azure Durable Functions to unify the load generation and event consumption across all scenarios. All three scenarios (FSF-K8S, ADF-Default, ADF-Netherite) consume events from the same Kafka topic, but each was assigned to a distinct consumer group. When a distinct consumer group is assigned to scenarios, each consumer group maintains a different offset and its own isolated view of the topic. This was critical for our experiments since it allowed isolated and parallel consumption for each scenario, which in turn guaranteed that each system consumed the same events. Figure 3.2 shows the workload generation. Checkout events are produced by a Python workload generator to Kafka and consumed independently by Flink StateFun and Azure Durable Functions (Netherite and Default), each using separate Kafka consumer groups.

To benchmark the system's behavior and performance under increased load, we generated the checkout events to the Kafka topic as follows:

- **Single-node setups (EP1 × 1)**: 100, 500 and 1,000 events
- **Single-node setups (EP3 × 1)**: 100, 500 and 1,000 events

3.6 Metrics and Data Collection

To conduct a meaningful comparison between scenarios, we captured the system logs at specific times in the workflow logic, such as checkout start and receipt generation times. The main metrics to evaluate the performance of each scenario were end-to-end latency and processing throughput.

3.6.1 Collected Metrics

- **Experimental Metadata:** `requestId`, `clientTimestamp`, and `experimentId` to associate logs with specific scenario.

- **End-to-End Latency:** In our benchmarks, end-to-end latency refers to the total time elapsed from the time the client creates the checkout event until the receipt is created for that specific event:

$$\text{Latency} = T_{\text{end}} - T_{\text{start}}$$

  Where:

  - T_{start}: The `clientTimestamp` is generated inside the Python workload generator and added as a property to the Kafka message payload.
  - T_{end}: The timestamp that we mark at the last step of checkout logic. To capture this, the receipt generation function logs a " GenerateReceipt" event with the `requestId`.

- **Average Throughput:** In our benchmarking scenario, average throughput represents the average number of checkout events processed per second by each system. To calculate the average throughput, we divide the number of checkout events to the total duration. The total duration is calculated by taking the time interval between the first generated checkout event by the client and the last processed event by the system:

$$\text{Throughput} = \frac{N}{T_{\text{receipt_last}} - T_{\text{client_first}}}$$

  Where:

  - N: Number of checkout events completed successfully.
  - $T_{\text{client_first}}$: The `clientTimestamp` of the first produced checkout event.
  - $T_{\text{receipt_last}}$: Timestamp of the last successfully processed checkout event by the system under test.

3.6.2 Apache Flink StateFun Logging

In the Flink StateFun scenario, since It was not managed by Azure, we configured manually logging library Log4j2 for structured logging within the function handlers. Logs were output directly to standard output, and then we collected them via Kubernetes logging mechanisms by manually saving to a file.

3.6.3 Azure Durable Functions Logging

For Azure Durable Functions, logging was configured through function `host.json` and monitored via Azure Application Insights:

- **Log Level:** Set to `Information` for detailed traces and disabled to sampling. When the sampling was active, it was causing some logs to be skipped, especially when systems were under a load.
- **Application Insights:** Provided by Azure out-of-box to capture logs and monitor the deployed Azure services. We enabled application insights and provided its API key as an environment variable to our Azure functions

3.7 Log Collection and Analysis

We exported the logs from both environments for offline analysis:

- Flink logs are manually collected using the `kubectl logs` command from the pod and saved to a file.
- ADF logs gathered using application insights by querying the traces table and exporting them as CSV files. Figure 3.3 shows an example query that we execute to get GenerateReceipt messages for each experiment.

After we gathered the log files, we analyzed and obtained meaningful metrics such as latency and throughput for each scenario, and visualized them as presented in the results section.

New Query 1*

Run | Time range : Last 24 hours | Show : 1000 results

```
1  traces
2  | where message has "GenerateReceipt"
3  | extend
4      userId = tostring(customDimensions.userId),
5      experimentId = tostring(customDimensions.experimentId),
6      clientTimestamp = tostring(customDimensions.clientTimestamp),
7      latencyMs = toint(extract(@"end_to_end_latency=(\d+)ms", 1, message))
8  | project
9      timestamp,
```

Results | Chart

timestamp [UTC]	userId	experimentId	clientTimestamp
7/12/2025, 9:38:45.125 AM	9ec33d65-669c-4542-953f-a35f...	FINAL_17_FINAL_EP1_1_1000_8	1752313120631
7/12/2025, 9:38:45.038 AM	0083468c-5814-40b1-ada5-96...	FINAL_17_FINAL_EP1_1_1000_8	1752313120631
7/12/2025, 9:38:45.036 AM	e588ff77-203c-4b71-9f89-c0a2...	FINAL_17_FINAL_EP1_1_1000_8	1752313120630
7/12/2025, 9:38:45.010 AM	38d1772a-cf84-4ef3-ac7a-0827...	FINAL_17_FINAL_EP1_1_1000_8	1752313120631
7/12/2025, 9:38:45.007 AM	0b23eb8b-aeb5-4423-ad6f-c17...	FINAL_17_FINAL_EP1_1_1000_8	1752313120631

Figure 3.3: Example Application Insights query

Chapter 4

Results and Analysis

This section presents the experimental results obtained from benchmarking two stateful serverless platforms—Azure Durable Functions (ADF) and Apache Flink StateFun (FSF)—under various configurations. The main performance metrics used were **throughput** (events per second) and **end-to-end latency**. These results give information on each platform's suitability for stateful workflows and also their ability to scale under load.

4.1 Benchmarking Scenarios

We implemented the exact same scenario in both platforms to ensure that the results were comparable. The results we discuss in this section were collected under the following configurations:

- **ADF Default Backend**: EP1 and EP3 single-instance setups.
- **ADF Netherite Backend**: EP1 and EP3 single-instance setups.
- **FSF (StateFun on Kubernetes)**:
  - Single-node setup with 1 TaskManager and 1 function handler pod (EP1 equivalent).
  - Single-node setup with 1 TaskManager and 1 function handler pod (EP3 equivalent).

4.2 RQ1: Architectural and Functional Differences

To answer RQ1, we compare the platforms in terms of programming model, consistency guarantees, fault-tolerance mechanisms, and state management. Table 4.1 gives the summary of these features side-by-side.

- **Programming Model**: Describes the way developers define workflows and which architectural approach is used. Functions communicate with each other either via asynchronous message passing or through central services called orchestrators.
- **State Management and Concurrent Access**: Since serverless functions are short-lived, the application state is externalized to an external storage such as Azure Storage Table, FASTER, RocksDB, and handled by the system transparently from the developers. Concurrent access guarantee tells whether the system ensures consistency of the shared state when two functions try to access and modify the same state value.
- **Fault Tolerance and Recovery Model**: In this thesis, we refer to fault tolerance as the capability of a system to handle failures and recover from them, such as runtime exceptions and network issues. Both ADF and FSF provide fault tolerance guarantees.
- **Execution Guarantees** once the system failures happen, the system provides certain execution guarantees, such as at-least-once execution, and an exactly-once execution guarantee. At-least-once execution guarantee is a weaker guarantee compared to exactly once guarantee, and the system usually retries the failed executions until they succeed. In exactly-once execution guarantee, it is guaranteed by the system that each delivered event is processed only once, and developers do not need to apply the extra checks or logic inside their application logic to prevent duplicate executions.
- **Polyglot support**: Shows whether the systems support multiple programming languages. It brings the flexibility to developers to implement a workflow or a scenario in their familiar programming languages.

Table 4.1: Architectural Comparison of Azure Durable Functions (ADF) and Apache Flink StateFun (FSF)

Feature	**Azure Durable Functions (ADF)**	**Apache Flink StateFun (FSF)**
Programming Model	Orchestrator-based (workflow-as-code async/await [9])	Actor-based stateful functions. Event-driven message passing [37]
Concurrent Access	Sequential access for requests that come from the same orchestrator and support entity locking mechanisms in C#	Sequential access to private state
Fault Tolerance and Recovery Model	Replay-based state recovery from history table[9]	State recovery from latest available checkpoint saved on external storage (e.g., S3, HDFS) [35]
Processing Guarantee	Exactly-once processing	Exactly-once processing
State Management	*Default:* Azure Tables, *Netherite:* FASTER key-value store [14]	RocksDB
Polyglot Support	Yes [10] (C#, Python, Javascript, Java, PowerShell etc.)	Yes [40] (Java, Python, Javascript, Golang)
Deployment Model	Fully Managed and offered as a service on Azure	Requires Kubernetes Cluster management Function handlers can deployed as FaaS [32]

4.3 RQ2: Performance Comparison Under Identical Workload

To compare the performance under identical workloads, we evaluated and compared throughput and latency metrics in different configurations for the checkout scenario, which was mentioned earlier in Section 3.1. We generated the checkout events with an increasing number of events: 100, 500, and 1000. To be able to compare the platforms under identical load, we made each platform consume the same Kafka topic through isolated consumer groups.

4.3.1 Throughput Results

4.3.2 EP1 Single Instance Comparison

An EP1 instance has 1 vCPU and 3.5 GB of memory. This experiment setup is the most basic premium plan (EP1) available for Azure Functions. The following analysis gives information about the performance of the systems under test when constrained to this single-node EP1 environment.

As shown in the Figure 4.1, Apache Flink StateFun (FSF-K8S) achieved

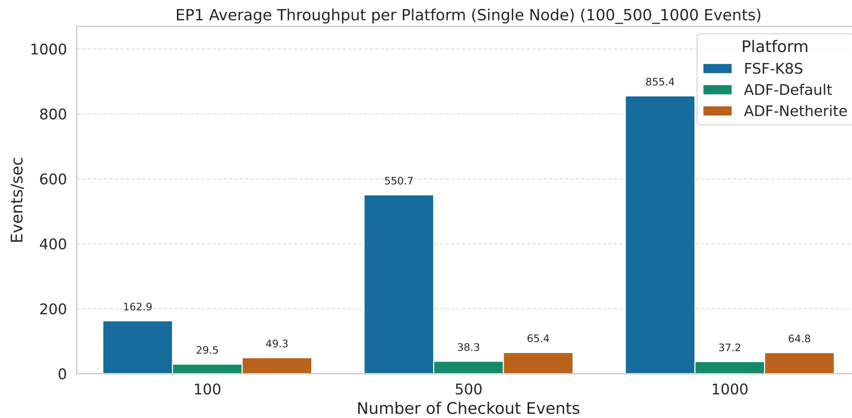


Figure 4.1: Average Throughput on EP1 (Single Instance)

the highest throughput across all event counts. FSF could process 100 and 500 checkout events under a second, with throughputs of 162.87 events/sec for 100 events and 550.7 events/sec for 500 events. These results show that FSF's architecture (underlying stream processing engine and efficient state management) can handle high volume of checkout events even on limited compute resources, EP1. On the other hand, Azure Durable Functions (ADF) shows significantly lower throughput compared to FSF. The Default backend configuration achieves a throughput of 38.27 events/sec at 500 events and shows a slight decrease for 1000 messages (37.25 events/sec). This is likely because the ADF Default backend has excessive storage access for state and orchestration history. ADF with the Netherite backend configuration performs better than the Default. It achieves almost double the Default backend throughput in all tested event counts. This shows Netherite's optimization for storage interactions and performance.

4.3.3 EP3 Single Instance Comparison

To test the impact of increasing compute resources on performance, we switched to another premium plan and used identical experimental methodology. The EP3 plan offers 4 CPUs and 14 GB of memory per instance. This setup allows us to evaluate whether low throughput results are due to compute resources or architectural limitations, especially in ADF configurations.

All platforms benefited from the increased CPU and memory resources and achieved better throughputs than EP1, especially the ADF Netherite

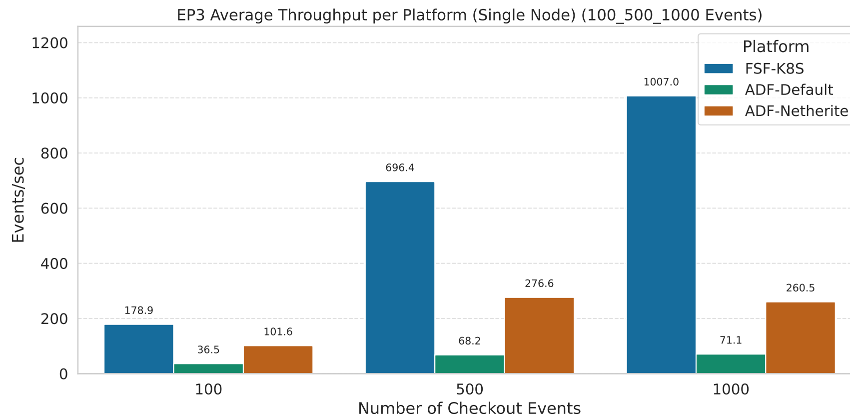


Figure 4.2: Average Throughput on EP3 (Single Instance)

backend. Figure 4.2 shows the throughput improvements on EP3 experiments. FSF continues as the leader in all tested event count, with the highest throughput, processing 1000 events in a second (1007 events/sec). ADF with Netherite backend shows a dramatic improvement in throughput compared to its single CPU EP1 performance. ADF Netherite achieves 260.47 events/sec for 1000 events, which indicates that Netherite can effectively benefit from increased CPU resources and parallelism. ADF with the Default backend configuration also achieves better result processing, with 71.07 events/sec for 1000 events. While it shows better performance compared to EP1, its throughput still remains the lowest compared to the Netherite and FSF, which indicates that for the ADF Default backend, storage access remains a limiting factor even with more compute resources.

4.3.4 Latency Performance (EP1)

Table 4.2 shows the detailed latency metrics for the EP1 experiments with mean and latency percentiles. Figure 4.3a shows the latency distribution for EP1 for all the tested event counts and the tested system. As shown in the figure, FSF-K8s demonstrates the lower latencies across all event counts. At the highest generated event count (1000 events), its mean latency is 935 ms. More importantly, its latency shows lower variance and a tighter range which indicates a stable performance with less outliers.

For ADF configurations, latency values are considerably higher and increase sharply when the number of events grows. The Default backend shows the highest tail latency, with its P99 latency exceeding 26s for

Table 4.2: Latency (ms) Summary for EP1 Single-Node Experiments (Mean, P90, P95, P99)

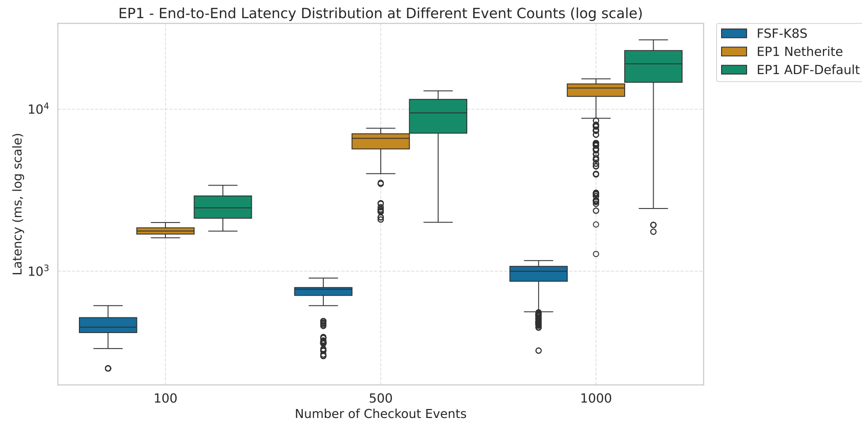
System/Backend	Events	Mean (ms)	P90 (ms)	P95 (ms)	P99 (ms)
Flink K8s	100	450	534	573	613
Flink K8s	500	726	803	820	884
Flink K8s	1000	935	1 084	1 094	1 104
ADF Default	100	2 483	3 151	3 254	3 368
ADF Default	500	9 014	12 420	12 693	12 918
ADF Default	1000	17 914	25 201	25 958	26 598
ADF Netherite	100	1 792	1 958	1 980	1 997
ADF Netherite	500	6 168	7 380	7 460	7 570
ADF Netherite	1000	12 308	14 944	15 152	15 301

1000 events. One reason for this high latency might be the excessive storage access. In our implemented checkout scenario, one checkout event causes 15 internal events and database inserts for the ADF default, affecting performance. Similarly, ADF with the Netherite backend configuration experiences increasing latency with increasing load; however, it shows better results over the Default configuration with a mean latency of 12308 ms for 1000 messages. The higher latencies for ADF compared to FSF are attributable to the architectural differences and deployment of the function handlers. Unlike ADF, FSF does not require a central coordinator for communication between functions, and it co-locates the state with the functions, which reduces the external I/O calls. Also, another factor might be the deployment of the FSF. Since FSF function handlers were deployed in the same Kubernetes cluster with Kafka, reduced network latency might also contribute to the lower end-to-end latency of FSF.

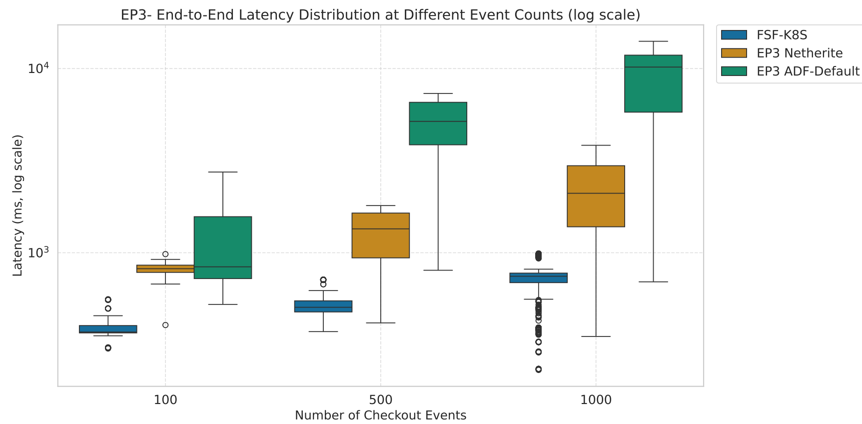
4.3.5 Latency Performance (EP3)

In the EP3 setup, FSF maintains the lowest latency with further improvements compared to EP1. All systems under test benefit from resource increases; however, ADF Netherite shows substantial improvement in latency in the EP3 plan. Its mean latency for 1000 events reduced almost 6x compared to EP1. This improvement, as we mentioned in throughput analysis, shows the Netherite can benefit from higher CPU and parallel processing and mitigate the latency bottleneck we observed in the single-CPU EP1 experiments.

ADF with the Default backend configuration still suffers from the highest



(a) Latency Distribution on EP1



(b) Latency Distribution on EP3

Figure 4.3: Latency distribution comparison for EP1 and EP3 setups.

latency. While adding more CPU and memory helps process events faster, the I/O bounded storage access acts as a bottleneck for the default backend. Table 4.3 summarizes the detailed latency metrics for the EP3 single-node experiments.

We plotted the latency trend for 1000 events to understand the platform behavior better. As shown in Figure 4.4, the ADF Default backend shows a step-like increase with a steep line when the request index increases. This shows the potential bottleneck or queueing behavior for ADF Default. In contrast, the ADF Netherite and FSF show more consistent and flatter latency trends.

Table 4.3: Latency (ms) Summary for EP3 Single-Node Experiments (Mean, P90, P95, P99)

System/Backend	Events	Mean (ms)	P90 (ms)	P95 (ms)	P99 (ms)
Flink K8s	100	402	555	556	557
Flink K8s	500	521	615	618	713
Flink K8s	1000	716	808	958	984
ADF Default	100	1 044	1 652	1 676	2 713
ADF Default	500	4 918	7 061	7 191	7 299
ADF Default	1000	8 769	12 589	12 935	13 810
ADF Netherite	100	816	906	916	920
ADF Netherite	500	1 273	1 746	1 756	1 795
ADF Netherite	1000	2 125	3 310	3 697	3 752

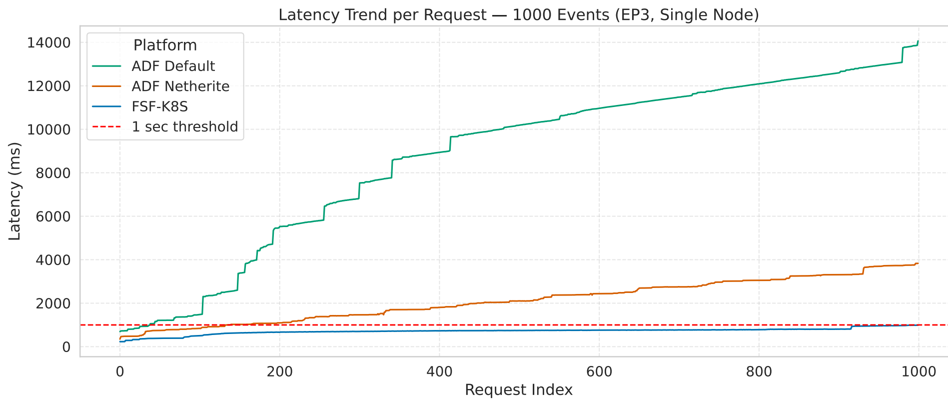


Figure 4.4: Latency trend on EP3 - 1000 Events

4.3.6 Comparative Analysis: EP1 vs. EP3 Performance

To better analyze the performance gain between different experiment setups, EP1 (1 CPU) and EP3 (4 CPU), we plotted the mean latency trend for the number of events for each platform on the same graph as illustrated in Figure 4.5. The dashed lines represent the EP3 setup, and the solid lines represent the EP1 setup.

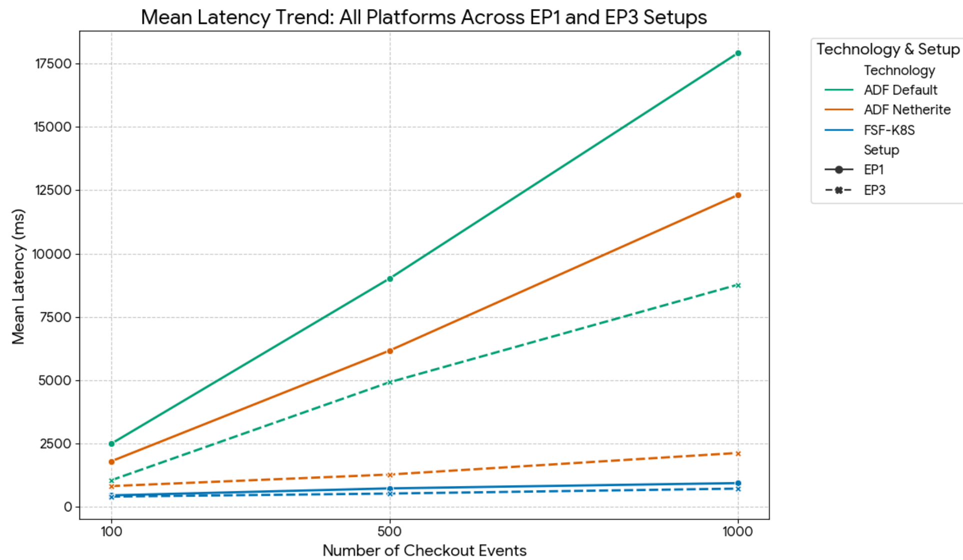


Figure 4.5: Mean Latency Trend of All Platforms in EP1 (1 CPU) and EP3 (4 CPU) setups. Solid lines represent EP1 performance, and dashed lines represent EP3 performance.

4.3.7 Overall Trend of Latency Reduction

As we can observe from the Figure 4.5, all platforms showed a consistent reduction in end-to-end latency in EP3 experiments. For each platform, the dashed lines (EP3) lie below the solid lines (EP1). The reduction is more clear in ADF Netherite which is shown with a dashed orange line. Netherite shows considerably lower values and a much flatter line compared to EP1, which stops at 2125 ms for 1000 events.

4.3.8 Apache Flink StateFun (K8s)

FSF achieves better results when we adjust resources from EP1 to EP3 equivalent. Its throughput increased from 855.43 events/sec to 1007.05 events/sec, which is roughly a %17 increase in throughput. Although we increased the CPU and memory almost 4x, the gain in the throughput was not linear.

- *Possible Reasons*: The non-linear but positive increase is likely because in EP3 experiments, only the resources of function handlers and task managers were increased, while other Flink specific configurations, such as task slots, parallelism, remained unchanged.

4.3.9 Azure Durable Functions (ADF-Netherite)

When we upgraded the EP3 plan, ADF Netherite backend gained significant performance improvements. Its throughput jumped from 64.82 events/sec (EP1) to 260.47 events/sec (EP3), which is almost a 4x linear increase in the throughput.

– *Possible Reasons*: This significant gain suggests that Netherite was constrained by a single CPU in EP1 experiments. Netherite uses a partitioned messaging system (by default 12 Event Hub Partitions) for its internal operations, so increasing the 4 CPU plan enabled Netherite to process more partitions concurrently, as a result, improved the parallelism and showed higher throughput

Chapter 5

Discussion

In this chapter, we present the key findings and overall results from the architectural analysis and the performance metrics. Following this, we propose future research directions and recommendations. Finally, we provide a personal reflection on the thesis journey.

5.1 Research results

Our experiments show that Apache Flink StateFun (FSF) consistently achieved the highest throughput and the lowest latency across all the tested experimental scenarios (EP1 and EP3). One key observation we obtained about FSF performance is that when we increase the load, FSF shows a lower variance in latency range, which demonstrates the stability of FSF under increasing load. These results are attributable to its architectural model, which uses the Flink stream-processing engine as an underlying framework. Also, FSF's actor-like model facilitates effective function communication and reduces external I/O calls by co-locating compute and state.

The ADF with Default backend configuration consistently showed the lowest throughput and the highest latency across our experiments. This modest performance is a consequence of the I/O-bound nature of the Default backend configuration. ADF with default backend configuration persists the orchestration history and instance state in Azure Storage Table excessively. In our implemented checkout flow, ADF inserts 15 database records in the history table for each checkout event. On the other hand, ADF Netherite backend showed significantly better results compared to the Default backend configuration, which is aligned with the results proposed in [14].

5.2 Suggestion for Developers

Based on these performance findings, for applications that have low latency requirements, choosing FSF can provide a better user experience. On the other hand, for applications that are less demanding, choosing ADF might suffice.

Regarding the architectural characteristics that we present in RQ1, both Azure Durable Functions and Apache Flink StateFun provide strong guarantees for stateful processing, such as exactly-once processing semantics, and an optimized state access mechanism. For applications that need to manage complex workflows that involve multiple steps and human intervention, such as process approval, an orchestrator-based solution like ADF can provide a better and more convenient programming model.

Beyond the objective architectural and performance metrics, our subjective experience during the implementation of the checkout scenario was as follows. The initial setup for Flink statefun was steep since it needs a manual Kubernetes deployment, which is not required in ADF. In terms of coding style, using async/await patterns felt more declarative while developing checkout orchestration logic in ADF.

5.3 Limitations

While this thesis provides a comparative analysis between the ADF and FSF, we encounter several limitations during our experiments. While we aimed to explore the scenario where we deployed remote function handlers to the Azure Functions (FSF-AzureFunc), this scenario was excluded from the presented results since Azure Functions Java has no support for non-blocking HTTP calls to the functions. While we could make a scenario up and running, it was a blocking function call that prevented the utilization of FSF capabilities. Thus, we excluded this scenario from the results and experiments for the fair comparison. Another limitation of our work is that we encounter issues with reliable log collection, especially with Azure Application Insights under high loads. In scenarios where we generated more than 5000 checkout events, logs were skipped, which limited our ability to analyze system behavior confidently.

5.4 Future Work

As we discussed in the results section, one of the benefits of SSS platforms is that they provide fault tolerance guarantees. One future research direction would be to discover the fault tolerance in more depth by introducing controlled failures on systems under test, such as function-timeouts, creating a scenario where the external storage is not accessible, and pod killing in the FSF scenario, then measuring how quickly they recover from these failures.

Another future direction would be the cost analysis of the systems under test. While ADF has various deployment plans, FSF is deployed on a Kubernetes cluster, which differs significantly and affects the cost directly. Investigating the function invocation patterns with different workloads would be used to identify under which conditions and workload characteristics one platform becomes more cost-effective than the other. Adding cost metrics together with latency and throughput metrics would provide a comprehensive comparison.

5.5 Reflection

Within the frames of this degree project, we have shown awareness of ethical aspects of research and development work, concerning methods, working methods, and results. Our work did not involve personal data; all data used during our experiments was randomized and synthetically generated, meaning there were no direct ethical considerations related to data privacy or sensitive information. Finally, this thesis contributes to United Nations (UN) Sustainable Development Goals (SDGs) by helping developers and system architects to make more informed decisions. It supports SDG 9: Industry, Innovation, and Infrastructure by helping develop resilient, sustainable, and more performant digital infrastructure and fostering innovation specifically in serverless cloud computing.

Chapter 6

Conclusions

In this thesis, we conducted a comparative analysis of stateful serverless architectures, where we specifically focused on two different architectural approaches: orchestrator-based and actor-like models. For this purpose, we chose ADF and FSF to compare their architectural differences and performance characteristics as defined by RQ1 and RQ2.

Our benchmarking results provided quantitative metrics about the performance characteristics of these systems. FSF showed superior results in terms of performance, and it achieved the highest throughput and the lowest latency in all the tested experimental scenarios. On the other hand, ADF with the default backend configuration showed modest performance and the highest latency, as a consequence of its I/O bounded nature for saving orchestration history to the Azure Storage Table. The ADF with Netherite backend configuration, however, showed significantly better performance improvements compared to the ADF Default backend. These findings directly answer RQ1, the performance attributes of the chosen systems.

Regarding the architectural analysis of these systems, our comparative analysis side by side showed the fundamental differences in their state persistence mechanism, state access patterns, and fault tolerance guarantees. These differences directly influence the application design and performance.

The findings and contributions of this thesis are valuable for system architects and developers as well as for the academic community. While RQ1 results give quantifiable and measurable data, RQ2 delivers a side-by-side comparison and trade-offs to understand the architecture and features of the systems. Also, our benchmarking scenario and methodology provide a groundwork for further tests and empirical analysis of these systems, as SSS are evolving and new architectural approaches are being proposed.

References

[1] A. F. M. Armbrust and R. Griffith, "Above the clouds: A berkeley view of cloud computing," *University of California, Berkeley, Tech. Rep. UCB*, 2009. doi: 10.1145/1721654.1721672

[2] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: State-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, pp. 7–18, 2010. doi: 10.1007/s13174-010-0007-6

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

[4] Amazon Web Services, Inc., "Serverless function, faas serverless - aws lambda," AWS, accessed: Jun. 9, 2025. [Online]. Available: https://aws.amazon.com/lambda/

[5] Google Cloud, "Cloud run functions," Google Cloud, accessed: Jun. 9, 2025. [Online]. Available: https://cloud.google.com/functions

[6] "Azure Functions – Serverless Functions in Computing | Microsoft Azure," accessed: Jun. 9, 2025. [Online]. Available: https://azure.microsoft.com/en-us/products/functions

[7] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," 12 2018. [Online]. Available: http://arxiv.org/abs/1812.03651

[8] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: State-of-the-art, challenges and opportunities," *IEEE Transactions*

on Services Computing, vol. 16, pp. 1522–1539, 3 2023. doi: 10.1109/TSC.2022.3166553

[9] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Durable functions: Semantics for stateful serverless," *Proceedings of the ACM on Programming Languages*, vol. 5, p. Art. No 133, 10 2021. doi: 10.1145/3485510

[10] cgillum, "Durable Functions Overview - Azure," accessed: Jun. 12, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview

[11] "cloudstateio/cloudstate," Jul. 2025, accessed on July 7, 2025. [Online]. Available: https://github.com/cloudstateio/cloudstate

[12] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful function-sasaservice," *Proc. VLDB Endow.*, vol. 13, pp. 2438–2452, 7 2020. doi: 10.14778/3407790.3407836

[13] "Apache Flink Stateful Functions." [Online]. Available: https://nightlies.apache.org/flink/flink-statefun-docs-release-3.2/

[14] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. S. Meiklejohn, and X. Zhu, "Netherite: Efficient execution of serverless workflows," *Proc. VLDB Endow.*, vol. 15, pp. 1591–1604, 4 2022. doi: 10.14778/3529337.3529344

[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, 2010. doi: 10.1145/1807128.1807152 p. 143–154.

[16] Datadog, "The State of Serverless," Aug. 2023, accessed on July 8, 2025. [Online]. Available: https://www.datadoghq.com/state-of-serverless/

[17] S. Kounev, N. Herbst, C. L. Abad, A. Iosup, I. Foster, P. Shenoy, O. Rana, and A. A. Chien, "Serverless computing: What it is, and what it is not?" *Communications of the ACM*, vol. 66, p. 80–92, 2023. doi: 10.1145/3587249

[18] "Serverless Computing," accessed: Jun. 6, 2025. [Online]. Available: https://aws.amazon.com/serverless/

[19] L. Ao, G. M. Voelker, L. Izhikevich, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18, 2018. doi: 10.1145/3267809.3267815 p. 263–274.

[20] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19, 2019. doi: 10.1145/3357223.3362711 p. 13–24.

[21] "Serverless Computing – AWS Lambda Pricing – Amazon Web Services," accessed: Jun. 10, 2025. [Online]. Available: https://aws.amazon.com/lambda/pricing/

[22] "Pricing - Functions | Microsoft Azure," accessed: Jun. 10, 2025. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/functions/

[23] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, "Formal foundations of serverless computing," *Proceedings of the ACM on Programming Languages*, vol. 3, p. Art. No 149, 10 2019. doi: 10.1145/3360575

[24] J. Wen, Z. Chen, X. Jin, and X. Liu, "Rise of the planet of serverless computing: A systematic review," *ACM Transactions on Software Engineering and Methodology*, vol. 32, p. Art. No 131, 7 2023. doi: 10.1145/3579643

[25] "Configure Lambda function timeout - AWS Lambda," accessed: Jun. 11, 2025. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-timeout.html

[26] ggailey777, "Azure Functions scale and hosting," accessed: Jun. 11, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale

[27] P. G. Lopez, M. Sanchez-Artigas, G. Paris, D. B. Pons, A. R. Ollobarren, and D. A. Pinto, "Comparison of faas orchestration systems," in *Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018*. Institute of Electrical and Electronics Engineers Inc., 7 2018. doi: 10.1109/UCC-Companion.2018.00049. ISBN 9781728103594 pp. 109–114.

[28] M. Azure, “Azure/durabletask,” Jul. 2025, accessed on July 16, 2025. [Online]. Available: https://github.com/Azure/durabletask

[29] “What is Step Functions? - AWS Step Functions,” accessed: Jun. 12, 2025. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html

[30] “Handling errors in Step Functions workflows - AWS Step Functions,” accessed: Jun. 12, 2025. [Online]. Available: https://docs.aws.amazon.com/step-functions/latest/dg/concepts-error-handling.html

[31] J. Spenger, P. Carbone, and P. Haller, *A Survey of Actor-Like Programming Models for Serverless Computing*, 2024, vol. 14360 LNCS, pp. 123–146.

[32] “Stateful Functions Internals: Behind the scenes of Stateful Serverless,” Oct. 2020, accessed: Jun. 13, 2025. [Online]. Available: https://flink.apache.org/2020/10/13/stateful-functions-internals-behind-the-scenes-of-stateful-serverless/

[33] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin, “Orleans: Distributed virtual actors for programmability and scalability,” Tech. Rep., March 2014. [Online]. Available: https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/

[34] cgillum, “Durable entities - Azure Functions,” accessed: Jun. 17, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities

[35] Apache Flink Community, “Fault tolerance,” Apache Flink Documentation, accessed: Jun. 27, 2025. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-master/docs/learn-flink/fault_tolerance/

[36] “Stateful Functions 2.0 - An Event-driven Database on Apache Flink,” Apr. 2020, accessed: Jun. 13, 2025. [Online]. Available: https://flink.apache.org/2020/04/07/stateful-functions-2.0-an-event-driven-database-on-apache-flink/

[37] “Architecture,” accessed: Jun. 14, 2025. [Online]. Available: https://nightlies.apache.org/flink/flink-statefun-docs-release-3.2/docs/concepts/distributed_architecture/

[38] cgillum, “Durable Functions storage providers - Azure,” accessed on July 20, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-storage-providers

[39] “apache/flink-statefun-playground,” Jun. 2025, accessed on July 7, 2025. [Online]. Available: https://github.com/apache/flink-statefun-playground

[40] “apache/flink-statefun,” May 2025, accessed: Jun. 27, 2025. [Online]. Available: https://github.com/apache/flink-statefun

TRITA – XXX-EX 2025:960
Stockholm, Sweden 2025

www.kth.se