Delft University of Technology

Masters Thesis

Anyone Can Cloud: Democratizing Cloud Application Programming

---

Author:
Wouter Zorgdrager

Supervisor:
Dr. Asterios Katsifodimos

Co-supervisor:
Dr. Marios Fragkoulis

Daily supervisor:
Kyriakos Psarakis MSc

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

in the

Web Information Systems Group
Software Technology

Student number: 4472977

Thesis committee:

Prof.dr.ir. G.J.P.M. Houben,	TU Delft, chair
Prof.dr. E. Visser,	TU Delft
Dr. A. Katsifodimos,	TU Delft, supervisor

An electronic version of this thesis is available at
https://repository.tudelft.nl/.

November 15, 2021

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

***Anyone Can Cloud*: Democratizing Cloud Application Programming**

by Wouter Zorgdrager

The cloud is widely adopted as a flexible and on-demand computing infrastructure. In recent years, a new and promising cloud paradigm emerged: serverless computing. Serverless computing promises a pay-as-you-go model and offers features such as autoscaling and high availability. Nevertheless, developing scalable cloud applications remains a painstaking task. Currently, programming models for the cloud mix operational code and business logic causing developers to spend a significant amount of time on other tasks rather than implementing the intended functionality. Moreover, the developer must consider distributed systems concerns such as consistency, communication, and persistence. Modern dataflow systems, such as Apache Flink and Google Dataflow, address these concerns but suffer from the same problem: they lack an intuitive programming interface for general-purpose applications. It remains an open problem to design a developer-friendly programming interface for implementing scalable cloud applications with strong guarantees.

In this thesis, we solve this problem by presenting an intuitive programming interface for scalable cloud applications in which developers primarily focus on business logic. Given a set of easy-to-follow code conventions, programmers author *stateful entities*, a programming abstraction embedded in Python. We present a compiler pipeline named StateFlow, to analyze the abstract syntax tree of a Python application and rewrite it into an intermediate representation based on stateful dataflow graphs. In addition, we present a set of building blocks that allow the execution of this intermediate representation on a target runtime system or cloud provider without a tight integration. Supported runtime systems include Apache Flink and Beam, AWS Lambda, Flink's Statefun, and Cloudburst, each providing a different set of guarantees. Finally, we introduce a client-side programming interface and HTTP server integration to interact with the deployed application.

We demonstrate that the execution with StateFlow typically incurs less than 1% overhead. Furthermore, we identify limitations of current dataflow systems in executing cloud applications at scale in a performance benchmark. Finally, we compare the expressiveness of StateFlow's programming abstraction to native runtime implementations. We show that StateFlow lets a developer write universal code that does not mix business with operational logic or the runtime's API and prevents vendor lock-in by allowing them to switch between runtimes in less than ten lines of code.

Acknowledgements

Finalizing my master thesis marks the end of my career at TU Delft. I must confess that doing this thesis was the most challenging but rewarding experience of all my time at TU Delft. This journey started six years ago when I began my bachelor's in Delft. In the second year, I got my first glimpse into the wondrous world of data-intensive systems when following the 'Big Data' course. Soon after, dr. Georgios Gousios asked me to support this course in subsequent editions and join the CodeFeedr project. In the years following, I got the opportunity to work with fascinating technologies such as dataflow systems and event streaming platforms. Later on, Georgios asked me to join the H2020 FASTEN project, where I created, maintained, and monitored a large-scale processing infrastructure. I learned so much in all these years. It feels fitting to finalize my degree with a thesis that combines all this acquired knowledge. I am proud of the final result. I hope that this thesis conveys a bit of my passion for scalable (cloud) applications.

There are many people I would like to acknowledge. First, I would like to thank Georgios for giving me many opportunities within the university. I enjoyed my time in the Software Analytics Lab and SERG while working on both CodeFeedr and FASTEN. Second, I want to thank my thesis supervisors: Asterios, Marios, and Kyriakos. It was always fun to talk with people who share the same research interests and brainstorm on new ideas. I want to thank Asterios for pitching the original idea, his endless enthusiasm, and for guiding me throughout the thesis process. Moreover, I'm grateful for Marios's constructive feedback and ideas during all the thesis meetings. I would like to thank Kyriakos for helping me with day-to-day struggles and being supportive at all times. I'm grateful for the help by Xander, who proofread my thesis document and provided me valuable feedback. I would like to thank the rest of my committee, prof. Houben and prof. Visser, for making time to evaluate my work.

Furthermore, I want to express my gratitude to friends and family. In particular, Mike and Joris ('de echte infoboys'), who were my study buddies throughout my time at university and are still good friends of mine. Attending lectures and working on assignments was way more fun with them. I want to thank my parents for everything they have done for me, their unconditional love and support, and for always encouraging me to work hard and follow my passions. I love you. Finally, I want to thank my boyfriend, Jelle, for always being there and making life more fun. Writing a thesis during the Covid times was not always easy. Still, he kept me sane, motivated me, and made the experience much more enjoyable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last ten years, the cloud revolutionized the way we deploy and administer software. Nowadays, two-third of all software spending for enterprise IT is based in the cloud Castro et al. [2019]. The last decade in cloud computing primarily focused on simplifying the configuration and management of computing infrastructure. The cloud promises a pay-as-you-go model where one only pays for the resources used and elasticity by providing almost infinite scaling with no up-front costs. This offering is also known as Infrastructure-as-a-Service (IaaS). A drawback of IaaS is that developers are left with the burden of scaling, which often results in overprovisioning of resources and increased costs Castro et al. [2019].

Consequently, a new paradigm for cloud deployment emerged: *serverless computing*. Serverless computing also adopts the pay-as-you-go model and provides automatic and unlimited up and downscaling of resources matching the demand. Although the serverless model is promising, modern cloud applications have to pay a high price for leveraging these cloud offerings Cheung et al. [2021]. Most prominently, developers struggle with cloud infrastructure abstractions, configuration, and deployment rather than dealing with application code Jonas et al. [2017]. In addition, developers are still responsible for implementing or integrating operational logic to support distributed systems concerns such as consistency, communication, and persistence. Inevitably, some of these cloud infrastructure concerns and operational logic leak through to the application layer resulting in more complex code.

At the same time, we witness a wide-scale adoption of modern dataflow systems such as Apache Flink Carbone et al. [2015], Google Dataflow Akidau et al. [2015], and Timely Dataflow Murray et al. [2013]. These systems do address concerns such as consistency, communication, and persistence while still being performant Akidau et al. [2015]. These guarantees and features make dataflow systems a proper execution model for cloud applications. However, their programming model focuses on distributed event processing, and they adopt a functional programming API that is not suitable for general-purpose applications.

To that extent, we identify two major problems we address in this thesis. First, the lack of a high-level programming model for cloud applications compatible with the serverless paradigm. We elaborate on this problem in Section 1.1. Second, even though dataflow systems are widely adopted as execution models, their functional programming models are far removed from the commonly used imperative programming model. In these systems, implementing general-purpose applications is practically infeasible. We highlight this problem in Section 1.2. These two open problems hinder developers in implementing scalable cloud applications with strong guarantees.

1.1 Programming for the Cloud

The most prominent serverless cloud solution is Function-as-a-Service (FaaS) Schleier-Smith et al. [2021]. Cloud providers offer these in the form of cloud functions: AWS Lambda, Google Cloud Functions, and Azure Functions. One of the disadvantages of Function-as-a-Service is the lack of application state. Support for this required the use of external services like DynamoDB. Moreover, developers are responsible for the integration of both services. To overcome these limitations, a new breed of systems arrived named Stateful Function-as-a-Service. These SFaaS solutions *do* support and integrate state into their execution and programming model. Besides the cloud providers, there are many industry and research (S)FaaS solutions such as CloudBurst Sreekanti et al. [2020], Beldi Zhang et al. [2020], and Flink Statefun[1].

However, most FaaS and SFaaS solutions still offer a limited, event-driven, and low-level programming interface Cheung et al. [2021]; Schleier-Smith et al. [2021]; Hellerstein et al. [2019a]. In addition, developers are still responsible for non-application logic such as serialization, state manipulation and function-to-function communication. As a motivating example, we show a simple user counter application in a FaaS system. Figure 1.1 shows this implementation in AWS Lambda, one of the most popular cloud-offered FaaS solutions, and with DynamoDB as a storage engine. DynamoDB fits the serverless paradigm as it provides autoscaling.

```python
dynamodb = boto3.client("dynamodb")
table = dynamodb.Table("users")
def user_counter(event, context):
    msg = json.loads(event["payload"])
    username = msg["username"]
    try:
        response = table.get_item(Key={"username": username})
    except ClientError: # user does not exist yet
        user_count = {username: username, count: 0}
        table.put_item(Item=json.dumps(user_count))
    else:
        user_count = json.loads(response["Item"])
        user_count["count"] += 1
        table.put_item(Item=json.dumps(user_count))
```

Figure 1.1: Simple user count application in AWS Lambda. We highlight the business logic with pink.

Only a few lines of this application concern business logic, and most code revolves around operational components such as state management, serialization, and event handling. As business logic is intertwined with infrastructure code, changing this logic or switching infrastructure components, like the database, requires heavy refactoring. Even worse, to move a cloud application between cloud providers or systems is prohibitive due to significant differences in the underlying systems. Cloud consumers fear such a *vendor lock-in* reduces bargaining

[1] https://flink.apache.org/stateful-functions.html

power in negotiating prices for these cloud offerings Schleier-Smith et al. [2021]. We argue that developers should have control over operational logic such as serialization and state storage, but it should never be part of the application code. At the same time, we argue the need for stronger guarantees and features necessary for distributed applications in the execution of cloud applications without exposing this directly in the programming interface.

1.2 Programming for Dataflow Systems

We believe that the main reason for the wide adoption of the dataflow execution model is that it offers a suitable abstraction for data-parallel computations: as long as a computation can either be partitioned randomly (e.g., `maps`) or partitioned given a key (e.g., `equi joins`), dataflow graphs can be trivially parallelized. Namely, each execution unit (e.g., a CPU core) can take over a partition of the data and process it sequentially, avoiding race conditions.

Currently, state of the art dataflow systems such as Apache Flink Carbone et al. [2015], Spark Armbrust et al. [2018] and Jet Gencer et al. [2021] can process millions of events per second on a single core, with latency in the order of milliseconds, with an *exactly-once* processing guarantee, and high-availability Silvestre et al. [2021]. These systems are already mature: Google offers Dataflow and Amazon offers Kinesis, both fully-managed cloud services for data processing, where users simply upload their code and the cloud takes over its execution, autoscaling and availability. This shows that dataflow systems are a great fit for the *serverless* model. However, developers must currently use a functional programming model when implementing such an application. Not only is this programming model cumbersome to use, it also requires heavy rewrites of the imperative code that developers typically use for expressing application logic. As a result, a developer can only enjoy the benefits of dataflow systems at the cost of an expensive development process. It remains an open problem to facilitate an imperative programming model for dataflow systems.

1.3 StateFlow

To address the two problems of programming for the cloud and dataflow systems, we introduce StateFlow: a programming model, compiler pipeline, and intermediate representation (IR) that compiles object-oriented Python applications into stateful dataflow graphs and executes them on existing dataflow systems. We argue that a stateful dataflow graph as an intermediate representation conveys all aspects of object-oriented applications required for execution in a distributed setting. At the same time, such a stateful dataflow graph allows outsourcing painstaking operational concerns such as configuration, scalability, and fault tolerance to a target distributed runtime system. Although StateFlow focuses on dataflow systems, it does not limit the execution of these graphs on other distributed solutions such as (S)FaaS.

StateFlow offers a developer-friendly programming interface where one writes imperative code rather than dataflow programs. StateFlow compiles this application code to a stateful dataflow graph. The most significant advantage of using

an IR is the ability to replace specific operational components and switch between the underlying distributed systems without modifying the application code. Currently, the supported systems include Apache Flink, Flink's Statefun, Apache Beam, Amazon's Lambda Functions, and CloudBurst.

In this thesis, we take inspiration from the work by Helland [2016] which explores practical approaches for implementing large-scale systems. Helland argues that a scalable application consists of at least two layers — a scale-aware lower layer and a scale-agnostic upper layer. Figure 1.2 shows a visualization of such an application.

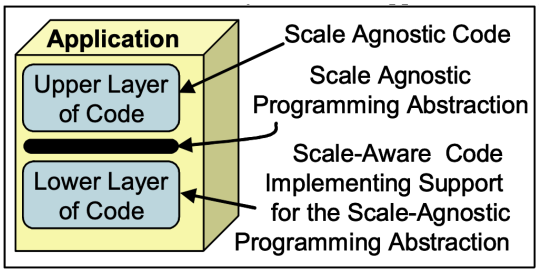


Figure 1.2: Scalable application with different layers. Figure taken from work by Helland [2016]

Furthermore, Helland introduces the concept of an entity: a single collection of data that a developer manipulates in the scale-agnostic upper-layer. These entities are addresable by their unique key, and the scale-aware lower-layer manages how to distribute them. According to Helland, most applications already have a design with an implicit form of such an entity. For example, customers, orders, shipments, or tax-payers Helland [2016]. StateFlow follows that line of thought and considers entities to be a proper programming abstraction for general-purpose, large-scale applications. In particular, StateFlow allows developers to implement applications using object-oriented code, where each class definition resembles such an entity.

StateFlow follows a similar architecture as presented in Figure 1.2. The *upper layer* corresponds to StateFlow's programming abstraction that compiles entities to a stateful dataflow graph. This pipeline follows the 'Lift and Support' approach, where a compiler *lifts* as much as possible to an intermediate representation and encapsulates what remains in user-defined functions Cheung et al. [2021]. In the abstraction of this upper layer, developers do not need to worry about the system's scalability. The *lower layer* represents the scale-aware system, like the dataflow or (S)FaaS system. StateFlow forms a bridge between this scale-agnostic and scale-aware layer using its IR.

Besides runtime functionality, StateFlow also offers a client-side model to interact with the deployed application. It uses its intermediate representation and integrates with event streaming platforms such as Apache Kafka. In this client-side model, developers write object-oriented code, and StateFlow handles event communication with the target runtime. Again, this programming model abstracts away from all operational aspects and allows developers to focus on business logic. Finally, StateFlow provides HTTP integrations and configuration for cloud deployments of both the runtime systems and the client interface. All in all,

StateFlow enables developers to implement, configure, and deploy end-to-end large-scale applications with minimal effort.

1.4 Research questions

In the previous section, we debated the need for a high-level programming interface that abstracts away from operational and infrastructural aspects. In addition, we opt for an intermediate representation in the form of a stateful dataflow graph. Therefore we design a pipeline that compiles such programs to an IR. This compiler compromises the first research question:

> **RQ1:** How does one transform object-oriented code to event-driven stateful dataflow graphs?

Moreover, we explore how a stateful dataflow graph fits the architecture of distributed systems and in particular dataflow systems. Additionally, we prefer loose coupling of the IR and the execution engine such that switching the engine is trivial which prevents a vendor lock-in. Therefore, we formulate the second research question:

> **RQ2:** Given a stateful dataflow graph, how does one execute this graph with loose coupling to an underlying distributed processing engine?

Finally, we explore the use of dataflow systems such as Apache Flink Carbone et al. [2015] as a universal execution engine for general-purpose applications. Notably, we explore how we benefit from the maturity of dataflow systems in terms of performance and guarantees. At the same time, we would like to identify the shortcomings of such systems concerning general-purpose applications. This leads to the third and final research question:

> **RQ3:** What is the performance and overhead of dataflow and (S)FaaS systems for general-purpose cloud applications, and what are the limiting factors?

1.5 Contributions

We summarize the contributions of this thesis as follows:

- We present a process for analyzing and transforming an object-oriented Python application to a stateful dataflow graph, and compare its expressiveness to native implementations on three different distributed systems: Apache Flink, Flink Statefun, and AWS Lambda.
- We desribe an intermediate representation (IR) for cloud applications and how that IR translates to a stateful dataflow execution graph. We demonstrate its interoperability by integrating it for a variety of popular distributed systems.

- We introduce a set of building blocks to ease integration with new target runtime systems. As a result, a developer can integrate the IR for new target runtimes in as little as 87-190 lines of code.
- We provide a set of integrations, deployment tools, and automatic configurations that allow a developer to deploy end-to-end applications. For example, we integrate StateFlow's entities as HTTP endpoints and we provide Kubernetes manifests to deploy runtime systems.
- We evaluate StateFlow's efficiency on each of the supported systems, reporting the overhead that StateFlow incurs on top of them. Additionally, we analyze the performance of these systems on the DeathStar benchmark, which simulates a realistic workload.

1.6 Thesis outline

We structure the rest of this thesis as follows. In the preliminaries, Chapter 2, we discuss some of the required background knowledge. Chapter 3 gives a high-level overview of StateFlow: a compiler pipeline and execution model for stateful dataflow graphs.

Chapter 4 details the programming model and compiler pipeline of StateFlow. Chapter 5 follows with a detailed explanation of how StateFlow executes stateful dataflow graphs and the several runtime architectures it supports.

In Chapter 6, we conduct a set of experiments, evaluating StateFlow from several angles: we show the expressiveness of the introduced programming model, the overhead that StateFlow, and, finally, we evaluate the complete system's performance using a benchmark.

Following the experimental results, we discuss these in-depth in Chapter 7. In addition, this chapter reviews the contributions and the related work of this thesis. We end this work with concluding remarks and a proposal for future work in Chapter 8.

Chapter 2

Preliminaries

In this chapter, we establish background knowledge and explain some of the concepts and terminology used in this thesis. In Section 2.1 we revise some programming language terminology and concepts with a focus on the Python language. This section lays a foundation for the compiler pipeline proposed in this work. Next, in Section 2.2 and 2.3, we introduce respectively dataflow systems and Function-as-a-Service (FaaS): two execution models that we build upon in this work. Finally, in Section 2.4 we introduce the practice of event streaming.

2.1 Programming languages

A *programming language* is a computer language compromising a set of instructions in the form of a syntax and is used to instruct a machine to perform specific tasks. In this thesis, we primarily work with Python: a popular high-level, versatile and general-purpose programming language. In the remainder of this section, we use Python syntax in the examples. However, most of the concepts and terminology generalize to other programming languages.

We often split a programming language into two components: **syntax** and **semantics**. The syntax describes the structure and the semantics describe the meaning of the programming language. For example, the following code is Python *syntax*:

```
5 + 10
```

The syntax of Python allows for the expression `<INT> + <INT>` where `<INT>` is an integer. On the other hand, the semantics of Python tell us that this syntactically valid expression performs the addition operation on both integers. Formally, the syntax of a programming language is defined by a *grammar*, similar to a natural language.

Functions and Classes Although the syntax of Python is much broader, we now highlight some of its elements and its terminology. In this work, we consider developers to define application functionality using classes and functions.

A **function**, procedure or method, is a block of reusable code encapsulating a single task. In Python, one defines a function by defining a function signature or header and the function body. A function signature compromises a **function name**, **formal parameters** and a **return type**. In the example below, the function name is `add`, its formal parameters are `x: int` and `y: int` and the return type

is `int`. In Python, it is optional to specify types (including the return type) for a function signature.

```
def add(x: int, y: int) → int:
    return x + y
```

Whenever we call or invoke this function: `add(5, 10)`, we label `5` and `10` as the **actual parameters**. Alternatively, the actual parameters are labelled **arguments** and the formal parameters **parameters**. We follow that convention in this work.

A **class** is a blueprint for objects and bundles together functionality and data. One defines the functionality of a class in functions and the data in variables. The class functions operate on those data, which modifies the state of the object. In Python, one defines a class like this:

```
class User:
    def __init__(self, username: str):
        self.username = username
        self.balance = 0

    def update_balance(self, new_balance: int):
        self.balance = new_balance
```

By defining such a class, one creates a new type `User`. Instantiation of the class type returns a class object or **instance**. The functions defined in a class are **methods**. A special method of a class is the constructor: `def __init__()`. This method is invoked upon object instantiation. Each method passes a `self` variable, which represents the instance of that class. Attributes assigned to this `self` variable are **instance attributes**. Commonly, all instance attributes are declared in the constructor method.

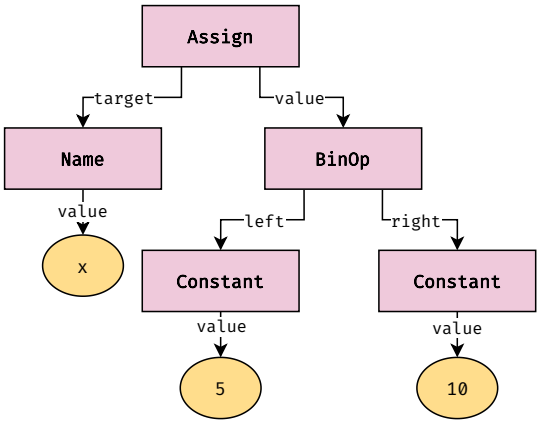
Types A data type is an attribute of data that instructs the compiler or the interpreter on the intended use of that data. Python is a dynamically typed language and does not enforce defining types for variables. Types are only checked at runtime and the type of a variable is allowed to change over time. On the other hand, Python does support **type hints** or **type annotations** in its syntax. For example: `x: int = 1`, defines the variable `x` with the type annotation `int`. These type annotations are not checked during compilation.

Abstract Syntax Tree A programming language is compiled or interpreted before execution on a machine. These compilers or interpreters commonly transform the program into an intermediate representation. Such a representation is useful for different kinds of analysis and optimizations before the machine code is generated. A common intermediate representation for compilers is an **abstract syntax tree** (in short AST). An abstract syntax tree is an ordered tree that represents the syntactic structure of the code according to the grammar of a language. This tree is 'abstract' as it does not denote every detail in the syntax. After the compiler executes several kinds of analysis and optimizations on the AST, it serves

as a base for (machine) code generation. In this work, we also use abstract syntax trees for static analysis and code transformations. Below we show an example of an AST. Consider the following piece of code:

```
x = 5 + 10
```

The corresponding AST looks like this:



On some occasions, we present an AST in its flattened version:

```
Assign(target=Name(value=x), value=BinOp(left=Constant(value=5),
                    right=Constant(value=10)))
```

Note, we sometimes merge AST nodes to simplify the visualizations.

2.2 Dataflow systems

Dataflow processing or *stream processing* is an execution model for the parallel execution of dataflow graphs. A **dataflow graph** or dataflow is a directed graph of nodes (i.e., operators) and edges where nodes represent computation, and edges represent how data flows from one node to another. Data enters the graph via sources — in the form of events — and leaves the graph again via sinks. We distinguish two kinds of operators in a dataflow graph, *stateless* and *stateful* operators. Stateless operators transform events without storing any information. Contrary, stateful operators do remember information from previously processed events (i.e., stateful operations). Commonly, stateful operators require events to have a key, which scopes the event to a particular piece of state. Dataflow systems allow processing of *unbounded data streams* entering the graph via the source nodes.

Dataflow systems execute dataflow graphs in a parallel and distributed manner. In general, stateless operators in the dataflow graph can be trivially parallelized by replicating the operation across multiple compute units. Similarly, stateful operators are partitioned and distributed across multiple machines or compute units. The dataflow system ensures that events with identical keys end up at the operator instance with the corresponding state. Finally, the dataflow

system moves events from one operator to another to simulate moving over the edges in the dataflow graph.

One such popular dataflow (or stream processing) system is Apache Flink [1]. We rely on Apache Flink in several contexts, and therefore, we use it as an example system throughout this work. On top of (stateful) dataflow graph processing, Apache Flink offers several features and guarantees. Most features and guarantees are not unique and generalize to other dataflow systems. Most notably, Flink allows processing large volumes of real-time data with low latency while also providing fault tolerance. Additionally, it provides strong consistency guarantees for both the state and the events. We highlight some of the processing guarantees in Subsection 5.3.2.

These dataflow systems often adopt a functional programming API in which developers define their program in the form of a dataflow graph. For example, a simple word count program in Apache Flink looks like the code presented in Figure 2.1.

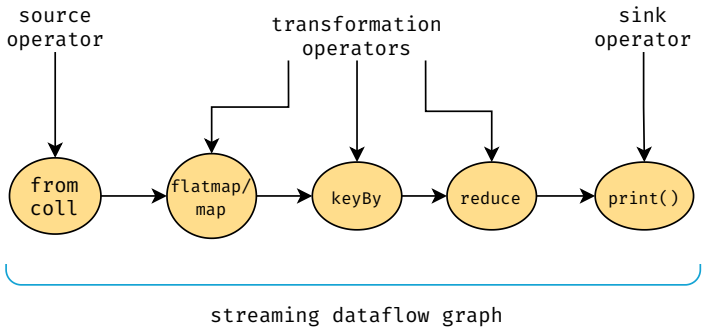
```
env = StreamExecutionEnvironment.get_execution_environment()

ds = env.from_collection(["the weather is always nice in Delft"])
ds = ds.flat_map(lambda line: line.split()) \
    .map(lambda i: (i, 1)) \
    .key_by(lambda i: i[0]) \
    .reduce(lambda i, j: (i[0], i[1] + j[1])) \
    .print()

env.execute()
```

Figure 2.1: Word count example using the Flink Python API (PyFlink). This example is adapted from the official Flink documentation.

The corresponding streaming dataflow graph:



In general, each transformation in the syntax corresponds to an operator in the dataflow graph. However, sometimes multiple operators are grouped together (i.e. `map` and `flatmap`). Data enters the dataflow graph via the source operator, in the example a list of strings. Via the edges, the data enters the different operators. Each operator performs a specific operation or transformation before forwarding

[1] https://flink.apache.org/

the data element to the next operator via the graph. Finally, the data enters its final destination in the form of a sink. In our case, the data is printed to the console, but this could also be another system such as a database.

2.3 Function-as-a-Service

Function-as-a-Service, or *FaaS*, is a category of services where one can define a function implementation and a service provider, like a cloud provider, deals with deployment, execution and management of this function. By using FaaS one can achieve a serverless architecture. Popular FaaS solutions include AWS Lambda [2], Google Cloud Functions [3], Azure functions [4] and OpenWhisk [5].

Advantages of using FaaS include, 1) less focus on the deployment of your application, 2) only pay for the resources you use, 3) scale up and down automatically and 4) enjoy all benefits from a cloud provider like geo replication. One the other hand, the concept of FaaS is rather new and lacks some important features. For example, dealing with state inside a function is not yet trivial. It requires interaction with another (database) service. Moreover, there are no guarantees with regard to fault tolerance. At best, functions are simply re-executed whenever it fails. FaaS solutions that do support application state out-of-the-box are labelled *Stateful Functions* or *Stateful Function-as-a-Service*. Popular SFaaS solutions include Flink Statefun [6] and CloudBurst Sreekanti et al. [2020].

2.4 Event streaming

An *event streaming platform* (ESP) is a distributed, scalable and durable system capturing streams of events from various sources. An ESP implements the publish/subscribe architecture and plays a significant role in the ingestion, storing, and processing of real-time event data. In this thesis, we mainly work with Apache Kafka, the most popular open-source ESP. We now explain some event streaming concepts in the light of Kafka.

Kafka has three main capabilities: 1) publish and subscribe to a stream of events continuously, 2) store event streams durably, and 3) process streams in real-time. One deploys Kafka as a distributed cluster, which stores all the event streams. With the help of a client, one ingests (publish) events into the cluster or reads (subscribe) from an event stream. Figure 2.2 shows all components in a Kafka setup.

Kafka stores event streams in the form of a *topic*. In addition, topics are partitioned to distribute load and make it scalable. A publisher (i.e., producer) writes to a Kafka topic, whereas a subscriber (i.e., consumer) reads from a topic. In this work, we use this pattern to interact between clients and runtimes. This approach has many advantages, and we highlight the most significant ones. First, event streams are durable, and Kafka provides delivery guarantees for all clients. Second, events streams are distributed, and Kafka supports distributed consumers.

[2] https://aws.amazon.com/lambda/
[3] https://cloud.google.com/functions
[4] https://azure.microsoft.com/en-us/services/functions
[5] https://openwhisk.apache.org/
[6] https://flink.apache.org/stateful-functions.html

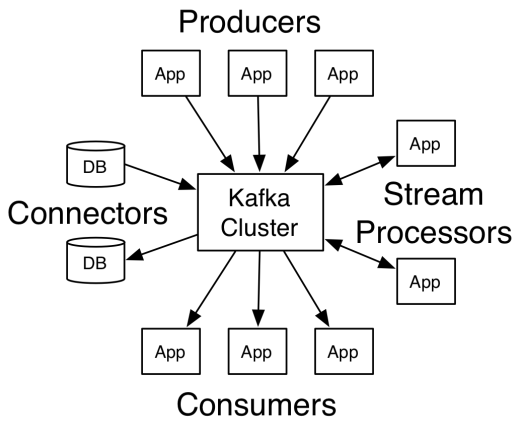


Figure 2.2: Components involved in a Kafka setup. Image taken from official Kafka documentation.

For example, multiple consumers (i.e., a distributed runtime) can read from the same topic, and events are distributed over the different consumers without event duplication or event losses. Finally, Kafka deals with *backpressure*: if downstream consumers cannot keep up with the incoming data from producers, Kafka acts as a buffer.

In this thesis, we use ESP's to interact between clients and runtimes. Rather than directly sending events from clients or runtimes (and the other way around), we send it via the ESP and enjoy all its features and guarantees. This work also covers AWS Kinesis: a service similar to that of Apache Kafka offered as-a-service by Amazon.

Delivery guarantees Undoubtedly, Kafka's consistency/delivery guarantee is its most significant feature. Kafka provides these guarantees in the complete pipeline from producer to consumer and even integrates with dataflow systems like Apache Flink. In the table below, we summarize each guarantee.

Guarantee	Description
At least once	This guarantees that messages are not getting lost, but may be duplicated.
At most once	This guarantees that message are never duplicated, but might be lost.
Exactly once	This semantic guarantees that a message is processed only exactly once.

Although these guarantees might sound straightforward, from an implementation perspective these are rather challenging. Therefore they pose as one of the key features of systems like Kafka and Flink. AWS Kinesis only supports the *at-least-once* guarantee.

Chapter 3

Introduction to StateFlow

In this chapter, we introduce StateFlow: a Python framework that simplifies the development and deployment of general-purpose cloud applications. StateFlow consists of two main components. First, it compromises a compiler pipeline that translates object-oriented Python code to a stateful dataflow graph. The second component accounts for executing these dataflows graph by porting to several target systems and providing a client-side interface. Both components are discussed in depth in respectively Chapter 4 and 5.

Using StateFlow, developers define classes that are compiled and deployed on a distributed runtime of choice. Since StateFlow uses an intermediate representation of the classes, and execution constructs for runtimes, adding new runtimes is relatively easy. At the same time, StateFlow offers a simple client-side interface to interact with the deployed application. This approach allows developers to write applications having large-scale requirements with minimal effort and an implementation close to plain Python code.

The name StateFlow is a contraction of the words *state* and *flow*. Both are important concepts in the world of distributed systems and stateful dataflow graphs. *State* is fundamental in stateful computation, and it is treated as a first-class citizen in dataflow systems, whereas *flow* captures the concept of data flowing through a distributed system.

All code from this thesis is open-source and can be found on GitHub [1].

3.1 Compiler Pipeline

The first component of StateFlow is a compiler pipeline which is shown in Figure 3.1. The main principle behind this compilation pipeline is to relieve the developer of the burden of distributed programming. At the base of this pipeline, developers annotate their Python classes with `@stateflow`. The annotation intercepts the class definition, parses the code into its Abstract Syntax Tree (AST), and triggers the compilation pipeline. StateFlow does not compile arbitrary Python code, instead it requires developers to define all functionality in class definitions. We argue that, in line with the work by Helland [2016], classes are a intuitive abstraction for large-scale systems.

In StateFlow we refer to class objects as *stateful entities*. Besides the class annotation, the framework assumes that developers type all method parameters and implement a `def __key__()` method. StateFlow uses this key method as a routing and translation mechanism to partition and distribute work among nodes

[1] https://github.com/delftdata/stateflow

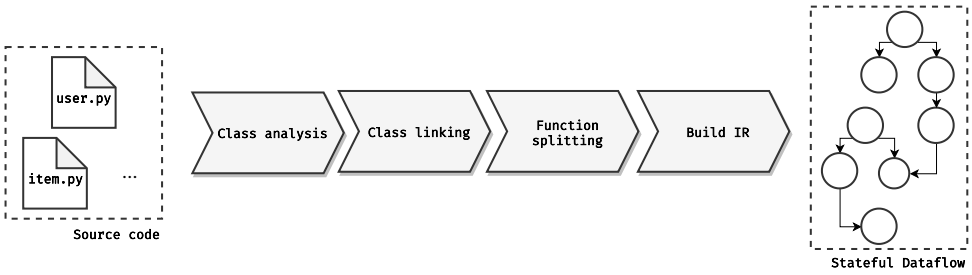


Figure 3.1: All stages of the compiler pipeline of StateFlow.

in a cluster. The key of a stateful entity cannot change throughout the entity's lifetime [2]. Several stages of static analysis and code transformation result in an intermediate representation for each annotated class. Chapter 4 discusses the pipeline components in detail.

For all source code analysis and AST transformations, StateFlow uses the LibCST framework [3]. LibCST parses Python source code into a Concrete Syntrax Tree (CST). Such CST encapsulates the Abstract Syntax Tree alongside its metadata like whitespace, parentheses, and comments. Using a simple API, LibCST allows StateFlow to traverse the AST of a piece of Python code efficiently. Moreover, it allows manipulation of this AST, which can be recompiled into Python code. LibCST uses its own intermediate AST/CST data structures, and therefore StateFlow supports a wide range of Python versions.

3.2 Classes to Dataflows

Most distributed systems, particularly dataflow systems, are event-driven, and therefore we require a translation from object-oriented code to event-driven execution. In Table 3.1, we show how the different object-oriented concepts translate to constructs in a dataflow:

Table 3.1: Translation from OO constructs to dataflows.

Python	**Dataflow**
Class	Operator
Object State	Operator State
Method Call Arguments	Event
Return Value	Event

Each class in Python translates to an operator (or vertex) in a dataflow graph. In these graphs, we cannot directly invoke operators similar to calling methods on an object. Instead, we send events into the dataflow graph, which ends up at the correct operator. The operator stores the *code* of a class and its actual *state* to reconstruct a *stateful entity*. Events encapsulate which method(s) to call and, finally, also store the return results.

[2] Currently, we do not have a static way of detecting key changes, and we rely on the responsibility of the programmer not to break this rule.

[3] https://github.com/Instagram/LibCST

More specifically, each dataflow operator is capable of executing all methods of a class, and it is triggered depending on the incoming event. Since operators can be partitioned across multiple cluster nodes, each partition stores a set of stateful entities indexed by the unique key of each individual entity. When a method of an entity is invoked, the entity's state is retrieved from the local operator state. Then, using the method's code that the operator stores, the method is executed using the arguments found in the calling event, as well as the state of the entity at the moment that the method is called. StateFlow offers routing functionality that uses the entity's unique key as well as the class name in order to direct the events to the right dataflow operator.

3.3 Executing Stateful Dataflow Graphs

The second component of StateFlow compromises a model to execute stateful dataflow graphs on top of several distributed runtimes, including dataflow systems. In addition, it provides a client-side interface to interact with these runtimes. StateFlow follows the *client-server model* and Figure 3.2 presents an overview of all its execution components.

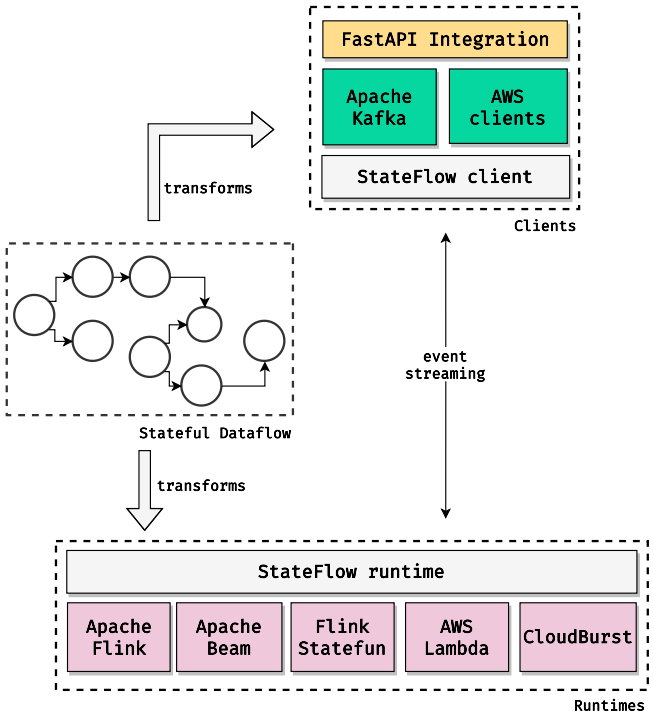


Figure 3.2: Overview of all execution components in StateFlow.

At this stage, both a client and a runtime rely on the intermediate representation generated by the compiler pipeline of StateFlow. StateFlow transforms and integrates this IR with several runtime systems, including, but not limited to, Apache Flink, Flink Statefun, and AWS Lambda. StateFlow provides a set of *building blocks* to compose architectures for each of these runtimes. Most prominently, these building blocks handle event routing and execution of methods in stateful entities. Using these building blocks has two significant advantages. First, event routing and execution are loosely coupled with the underlying runtime, and

therefore switching between runtimes is trivial. Secondly, it requires little code to integrate with runtime systems since StateFlow captures most functionality in its building blocks. StateFlow does not limit execution to dataflow systems like Apache Flink and supports FaaS systems like AWS Lambda. In general, StateFlow can support any distributed runtime as long as it offers some sort of storage or state.

On the client-side, StateFlow allows developers to interact with the defined classes like in native Python code. For example, one can construct instances and call methods. In the background, it sends and receives events from the runtimes using one of the clients (i.e., the Kafka client). The choice of a client depends on the selected runtime. For example, to use a Flink runtime requires the use of a Kafka client. On top of the client, one can enable an HTTP server using FastAPI. This integration creates HTTP endpoints, allowing one to interact with the stateful entities in the runtime via REST calls.

All details on the execution of stateful dataflow graphs, for both client and runtimes, are presented in Chapter 5.

3.4 Why Python?

When working with distributed systems, Python might not be the first programming language to come to mind. It is considered relatively slow, especially compared to more low-level languages like Java, Go, or C++. On the other hand, Python is a popular language [4] and relatively simple to learn. Moreover, we observe a shift in the adoption of Python support in dataflow and cloud systems. For example, Apache Flink started adding support for Python a few months ago besides its original interface in Java and Scala. In addition, all cloud providers offer Python support for their FaaS solutions.

3.5 Running Example

In this section, we present an example of two classes defined with StateFlow. The classes are shown in Listing 1 and 2. First, we define an `Item` class: This class encapsulates an item which has a `price` and a `stock`. The stock can be updated, and it can be verified if an item still has enough stock. The unique key of a stateful entity of the type `Item` equals its `itemid`. Secondly, we define an `User` class: This class encapsulates an user which has a `basket` and a `balance`. A user's balance can be updated, and items can be added to a user's basket. The unique key of a stateful entity is determined by the `userid`, similar to the `Item` type. Notice how the User class interacts with the Item.

[4] In a 2021 Stackoverflow survey (https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language) Python was the third most popular language behind Javascript and HTML.

```python
@stateflow
class Item:

    def __init__(self, price: int, itemid: int):
        self.price: int = price
        self.stock: int = 0
        self.itemid: int = itemid

    def __key__(self):
        return self.itemid

    def set_stock(self, stock: int):
        self.stock = stock

    def enough_stock() -> bool:
        return self.stock > 0
```

Listing 1: Code example: An *Item* class definition.

```python
@stateflow
class User:

    def __init__(self, userid: int):
        self.basket: List[Item] = []
        self.balance: int = 1000
        self.userid: int = userid

    def __key__(self):
        return self.userid

    def set_balance(self, balance: int):
        self.balance = balance

    def add_to_basket(self, items: List[Item]) -> bool:
        total_price: int = 0

        for item in items:
            if item.enough_stock():
                total_price += item.price

        if self.balance < total_price:
            return False

        self.basket = items
        return True
```

Listing 2: Code example: An *User* class definition.

We use these classes as a running example in subsequent chapters.

Chapter 4

Imperative Code to a Stateful Dataflow Graph

In this chapter, we elaborate on the compiler pipeline of StateFlow as introduced in Figure 3.1. The compiler pipeline allows developers to write object-oriented Python code, which StateFlow compiles to an intermediate representation. StateFlow uses and extends the original CPython compiler [1]. In general, Python source code (`.py`) is first *compiled* into to Python byte code (`.pyc`). This bytecode is then *executed* on a Python Virtual Machine. Although Python code is compiled into bytecode, code is not *built* and *linked* similarly to languages such as C++ and Java. Therefore, Python is considered to be an *interpreted* language Subasi [2020]. Figure 4.1 shows this default pipeline and how StateFlow's pipeline fits into this.

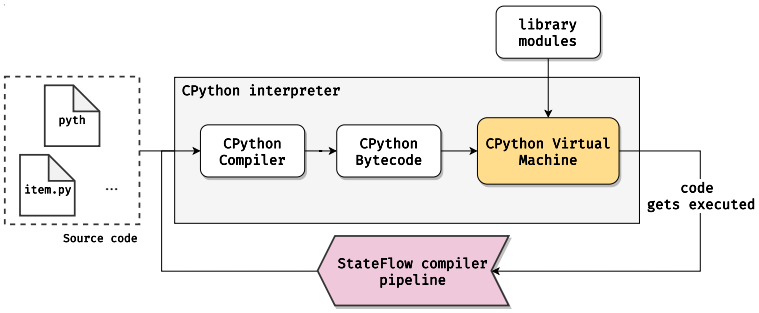


Figure 4.1: Full compilation pipeline including StateFlow.

In this pipeline, the original code is compiled and executed as usual. However, we intercept class definition and recompile these classes after they pass the StateFlow compiler pipeline. For each annotated class definition, we derive its AST, which we analyze and transform before its recompiled. Therefore, the complete source code is only compiled *once* whereas all class definitions are compiled *twice*. Due to the dynamic nature of Python, we recompile code during execution, and therefore, the developer only has to run the code once. Note, we do not modify the original CPython compiler nor use its internal representation.

StateFlow compilation pipeline consists of several stages, starting with a round of static analysis on all annotated classes explained in Section 4.1. Section 4.2 presents the second round of analysis, in which StateFlow links the classes that interact with one another using method calls. All methods with remote interaction

[1] Python has many compilers including: Stackless Python, Jython, CPython and PyPy. CPython is the default Python compiler and is widely used.

are a candidate for function splitting shown in Section 4.3. The splitting algorithm transforms code into the continuation-passing style form Reynolds [1972]. At the end of the pipeline, StateFlow builds an intermediate representation as explained in Section 4.4. Finally, we explain some of StateFlow's programming model limitations in Section 4.5.

4.1 Class Analysis

The class analysis is triggered by annotating a Python class with `@stateflow`. In Python, such an annotation is called a decorator: A metaprogramming technique to modify the underlying structure at runtime. This decorator is a function call and can also be called explicitly. For example, both code snippets are equivalent:

```python
@stateflow
class User:
    def __init__(self):
        ...
```

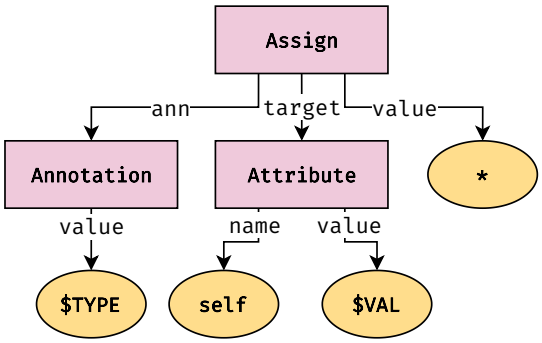
```python
class User:
    def __init__(self):
        ...
User = stateflow(User)
```

In this first round of analysis, we focus on individual classes. We infer the static properties of a class, like its attributes, by traversing each AST. More specifically, we collect 1) all instance attributes and their available types and 2) all methods and their parameter and return types. We identify instance attributes as assignments to `self` attributes somewhere in the class methods. Although defining instance attributes outside the class constructor is considered bad practice, StateFlow supports identifying these attributes in non-constructor methods. StateFlow does not support instance attribute definitions outside of the class definition.

Python is a dynamically typed language. Therefore, it does not require a developer to explicitly indicate data types. However, we encourage developers to use Python's type-hint system, allowing StateFlow to derive data types statically. Although encouraged in general, we enforce the use of types for method parameters. Annotating parameters is necessary as we need to statically derive how stateful entities interact. For example, the `def add_to_basket` method from the User class (Listing 2) interacts with a list of Item entities. Therefore, the `items` parameter needs to be typed as `List[Item]`. StateFlow supports all primitive types, lists, and class types. For class types, we distinguish between classes that are also stateful entities and those that are not. The latter could be auxiliary classes or classes from external libraries, and those will not be explicitly part of the stateful dataflow graph. StateFlow does not incur any limitation on using an external library in the code.

In order to collect these properties, we match on patterns in the AST. For example, to find a instance attribute we try to match the following AST node [2]:

[2] This is a simplified version of the actual AST node that is matched.



where * is a wildcard and $VAL, $TYPE are variables that are matched. For example, the statement `self.stock: int = 1`, with the corresponding AST node:

```
Assign(value= Integer(1),target=Attribute(name="self",
    value="stock"), ann= Annotation(value="int"))
```

matches this pattern and we derive the attribute name `$VAL=stock` and type `$TYPE=int`. In Python, `self` is not a reserved keyword, and using it is merely a convention. However, StateFlow relies on developers using this convention; otherwise, the pattern matching will not work. In this pass over the AST, StateFlow also verifies if the developers overrides the `def __key__()` method. This method should return a static result and cannot change during the lifetime of a stateful entity. There is no additional analysis to ensure that this method is static.

The analysis results in a `ClassDescriptor` which stores all these statically derived properties. The `ClassDescriptor` of the Item stateful entity can be found in Figure 4.2. We describe all instance attributes, `price`, `stock`, and `itemid` as state. Moreover, we store the input and output types for the methods `enough_stock` and `set_stock`. The input of a method is described as an ordered list of tuples with parameter names and types, whereas for the return variables, we keep track of only the types.

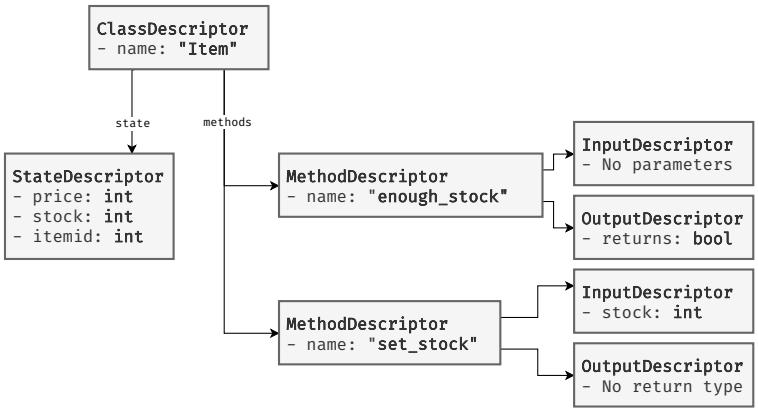


Figure 4.2: Analysis results for Item stateful entity.

4.2 Class Linking

In Section 4.1, we show how the first round of static analysis results in a description for each annotated Python class. In a second round, we link the methods

of all analyzed classes. That is, methods that have interaction with methods of other stateful entities. These methods are candidates for function splitting (Section 4.3). We cannot access the classes defined later in the code when interpreting and analyzing a class definition. Therefore, we can only link classes after the first round of class analysis. In this first round, we stored all analyzed classes in a global context. To trigger the second round of analysis, the developer has to execute: `stateflow.init()`.

In this second round, the `MethodDescriptor`'s of all analyzed classes are traversed. As mentioned before, StateFlow assumes that if a method interacts with another stateful entity, a reference to that entity is defined as parameter type. This type hint has been extracted by StateFlow in the first round and parsed to the correct type. As Python considers type hints to be an expression, we evaluate the AST of such an expression to make parsing easier. For example, the type hint `x: List[Item]` requires the import of the generic `List` and is parsed as:

```
Annotation(value=Subscript(value='List',
      slice=Index(value='Item')))
```

The evaluation of this type hint results in the original syntax: `List[Item]`. To this syntax, we apply a regular expression that parses into `Item`. With a lookup in the global context, we find and link to the `Item` entity.

4.3 Function Splitting

For simple functions that do not call other remote functions, execution is rather straightforward. If, for example, the method `User.add_to_basket` calls the (remote) method `item.enough_stock` whose state lies on a different partition, the situation becomes more complicated. The dataflow system cannot stop and wait for the remote function to complete and return before moving on with processing the next event. Instead, it must 'suspend' the execution of `add_to_basket` right at the spot that the remote function `item.enough_stock()` is called until the remote function is executed and an event comes back to the `User` operator with a return value.

In order to do this, we adopt a technique to transform the imperative functions into the continuation-passing style (CPS) Reynolds [1993]. More specifically, we propose an approach to split a function definition into multiple functions at the AST level. In the running example, the only candidate eligible for function splitting is the method `def add_to_basket(self, items: List[Item])`. None of the other methods have any reference to stateful entities. Notice that we label this technique as 'function splitting' although we apply it to class methods. We explain 'function splitting' in the following subsections based on the Python language and the Python AST grammar. However, we argue that many of the explained techniques can be applied to other (object-oriented) programming languages.

4.3.1 Simple splitting

To introduce the idea of the function splitting we start with a simple example:

```
1 def buy_item(self, amount: int, item: Item):
2     total_price = amount * item.price
3     is_removed = item.remove_stock(amount)
4
5     return total_price
```

This method could be part of the User stateful entity. As we execute this piece of code, we encounter a problem while invoking `item.remove_stock(amount)`. The item entity is located at a different operator in the dataflow graph and it cannot be invoked without suspending execution of the current method. Therefore, we split the function at line 3 and form two new functions:

```
1  def buy_item_0(self, amount: int, item: Item):
2          total_price = amount * item.price
3          remove_stock_arg = amount
4          return amount, item, total_price,
5                                {"_type": "InvokeMethod",
6                                "args": [remove_stock_arg],
7                                ..}
8
9  def buy_item_1(self, total_price, remove_stock_return):
10     is_removed = remove_stock_return
11     return total_price
```

Now, `buy_item_0` evaluates the 'first part' of `buy_item` (line 2), evaluates the parameters of the remote call (line 3) and finally returns (line 4-7). In this return statement, local variable definitions (`total_price`) and metadata for the remote call is returned. This is necessary to invoke the remote call and to continue with the execution afterward. Figure 4.3 shows how the call graph changed *before* and *after* the split. After splitting the functions into multiple, we can safely move back and forth between dataflow operators to call remote functions. By returning and storing the local state of an intermediate function, we ensure 'continuation' when the next function is invoked. This approach allows suspending computation in between dataflow operators, which fits the dataflow execution model.

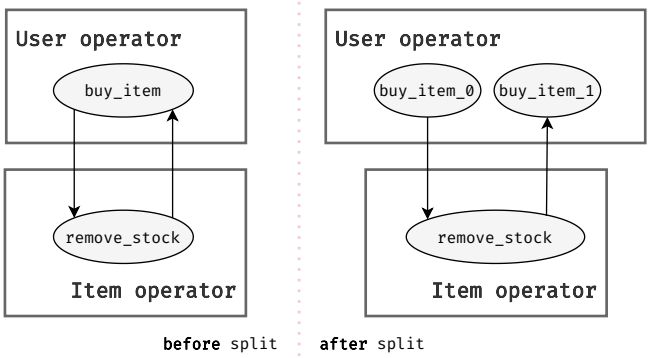


Figure 4.3: Call graph *before* and *after* the function split of `buy_item`.

Now that we have shown an example of a simple split, we attempt formalizing the splitting algorithm. As shown in the previous examples, StateFlow performs all code transformations by transforming the AST of the class and function definitions. We now present a set of algorithms in the form of visualizations of the AST nodes. The algorithms both transform and analyze these AST nodes and we apply the algorithms to each class method. For simplification, details, and edge cases have been omitted in these algorithms. For the visualization of the AST nodes, we use a minified version of the LibCST specification, close to the original abstract Python grammar [3]. All algorithms have access to a 'global context' that stores all previously analyzed stateful entities and provides utility functions. These utility functions and its descriptions are presented in Table 4.1.

Table 4.1: Auxiliary functions of the global context.

Name	**Description**
`callToReturn`	Transform a `Call` node to an expression which encapsulates the metadata of that call. This instructs the execution model to execute the remote call.
`children`	Given an AST node, it returns a list of all child nodes. In the context of AST's, these child nodes are ordered the same way as defined in the code.
`buildReturnId`	Builds an identifier for the return variable of a Call node. For example, a call to the method `remove_stock(amount)` will have the return value `remove_stock_return`.

Identify remote call First, we define an algorithm `IsRemoteCall` to identify if an AST node is a call to a remote function (Figure 4.4). This algorithm accepts an AST node, a list of typed declarations, and the global context. The typed declarations are an ordered list of variable names and their types.The order of this list is equivalent to the order in which these variables are declared. For example, consider the following lists of statements:

```
1 x, y = 0
2 y: int
3 p: str = ""
4 x: Item
```

After analyzing all statements of this code snippet, the typed declarations are: `((x, None), (y, None), (y, int), (p, str), (x, Item))`. The `IsRemoteCall` algorithm first verifies if the AST node matches the correct form (step 1). More specifically, we look for `Call` node with an `Attribute`. From this Call node, the variable and function names are extracted. The typed declarations are traversed in reverse order since we need to know the type of the *latest* declaration of a variable (step 2). If the variable name matches *and* the type of this variable is

[3]The AST grammar of Python is found on the documentation website: https://docs.python.org/3/library/ast.html

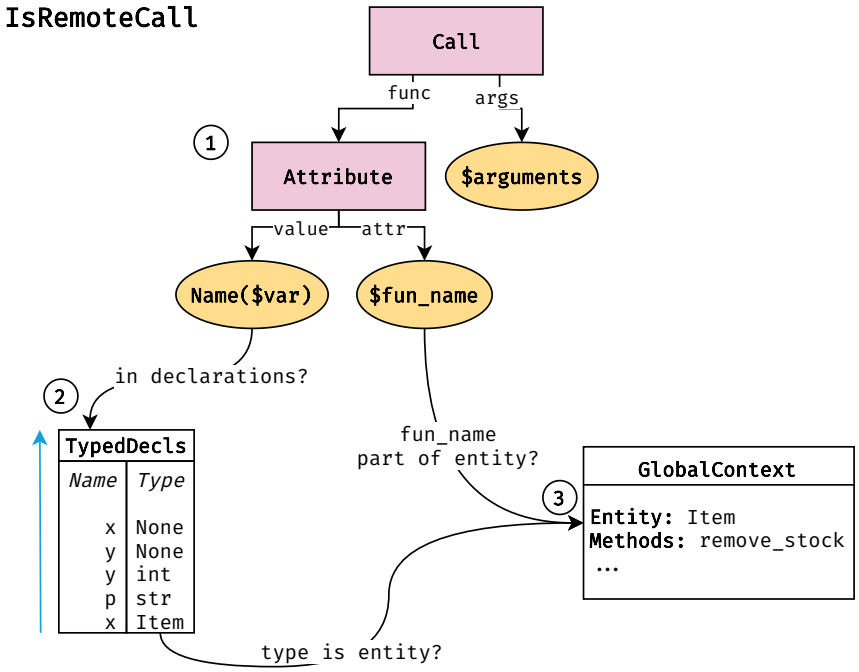


Figure 4.4: Visualization of the `IsRemoteCall` algorithm.

an (remote) entity (step 3), the algorithm returns *true*. In all other scenarios, the algorithm returns *false*.

Build function definition The second algorithm we define is `BuildFunctionDef` in Figure 4.5. This algorithm builds a new function definition, given a list of statements and an ordered list of usages and definitions. For example, the following code snippet:

```
1 x, y = 0
2 z = x + y
3 p * 4
```

has three statements corresponding to the three lines in the snippet. Note that a source code line is not always equivalent to a statement. A control flow element, like an if-statement, might span multiple lines. In the example, the ordered list of usages an definitions is `((x, Def), (y, Def), (x, Use), (y, Use), (z, Def), (p, Use))` The order of this list should be equal to the order of evaluation. In StateFlow we benefit from the LibCST AST implementation, which embeds the lexical order in its AST data structure. In general, it might require knowledge of the semantics of a programming language to decide on the order of this usage/definition list. For example, on line 2, first `x` is referenced followed by `y` before `z` is defined. We know this because the Python language first evaluates the right-hand side of an assignment statement [4].

[4] As documented on the official Python website: https://docs.python.org/3/reference/expressions.html#evaluation-order

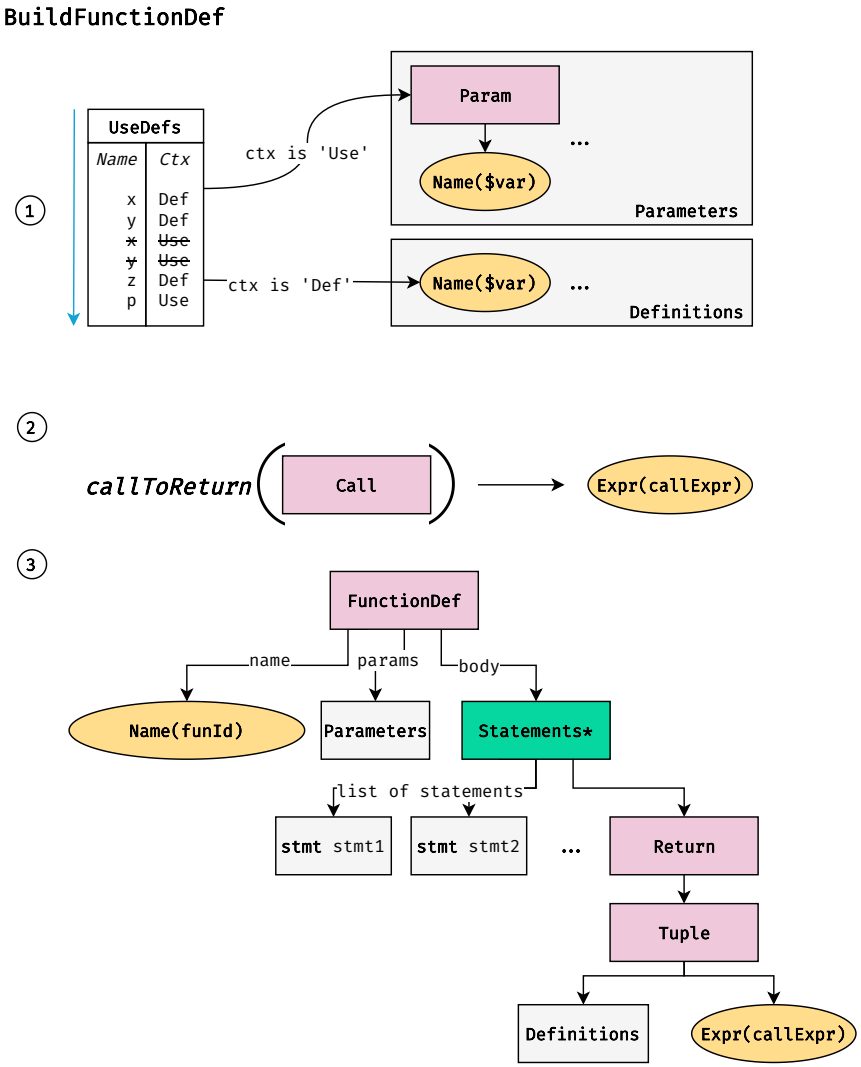


Figure 4.5: Visualization of the `BuildFunctionDef` algorithm.

In `BuildFunctionDef`, we iterate over these usages and definitions (Figure 4.8). A *definition* is generally added to the return statement, whereas a *used variable* is defined as a parameter (step 1). For 'Use' variables, we *only* add them as a parameter if it has not been defined before. For example, in the code snippet, `x` and `y` are used on line 2 but already defined on line 1. Therefore these variable names do not need to be part of the function parameter definition. The goal of adding these parameters and return variables is to ensure the *continuation* of the original function definition. In this context, a function parameter is a *dependency* on one of the previous function definitions, whereas the return variables are output on which subsequent functions can *depend*. If we split a function because of a remote call, metadata is appended to the return statement to instruct the execution model (step 2). We defined this operation as `callToReturn` in the global context. More specifically, given a call, (e.g. `item.remove_stock(amount)`) we build a Python dictionary in the following form:

```
{
  "_type": "InvokeMethod",
  "class_name": "Item", # Name of stateful entity
  "ref": item, # Reference to the entity variable
  "invoked_method": "remove_stock",
  "args": [amount]
}
```

We append this expression, a Python dictionary, as the last item in the return statement. Not all function definitions need to return this call metadata since not all function are followed by a remote call. For instance, in the first example `def buy_item_1`, does not need to return with instructions for a remote call. This metadata has static properties like the class name and the invoked method, whereas the reference to the entity and the call arguments are dynamically determined at runtime. In the end, we build a new function definition (step 3) with 1) the extracted parameters, 2) the input statements, and 3) a new return statement with potentially the call metadata.

Split a function Finally, we define an algorithm for splitting a function `SplitFunction` in Figure 4.6, 4.7, 4.8 and 4.9.

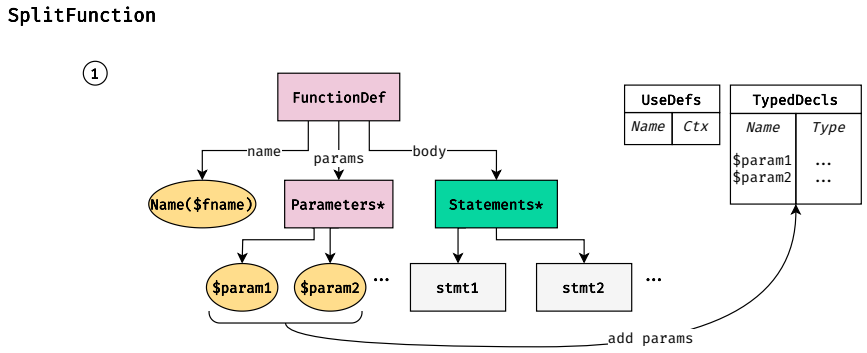


Figure 4.6: Visualization of step 1 of `SplitFunction` algorithm.

This algorithm expects a function definition as input alongside the global context. In the previous analysis phases, StateFlow marked this function definition eligible for the splitting algorithm. Initially, all parameters of the function definition are considered as typed *declarations* (Figure 4.6). Although this function definition does not explicitly declare these parameters, they ought to be available in the function scope as call arguments. In a previous AST analysis, we already enforced types to be available for the parameters.

Next, we iterate over all statements in the function definition (Figure 4.7). For each statement, we iterate over its child nodes. The order of child node traversal is identical to how it appears lexically in the code. We maintain a list of variable usages and (typed) definitions for the current list of statements, respectively `UseDefs` and `TypedDecls`. If a child node is a remote call, we go to step 3 to perform a 'split' (Figure 4.8). If a node is a `Name` or `AnnAssign`, we add these variables to `UseDefs` or `TypedDecls`. A statement is added to the current list of statements when all child nodes have been traversed.

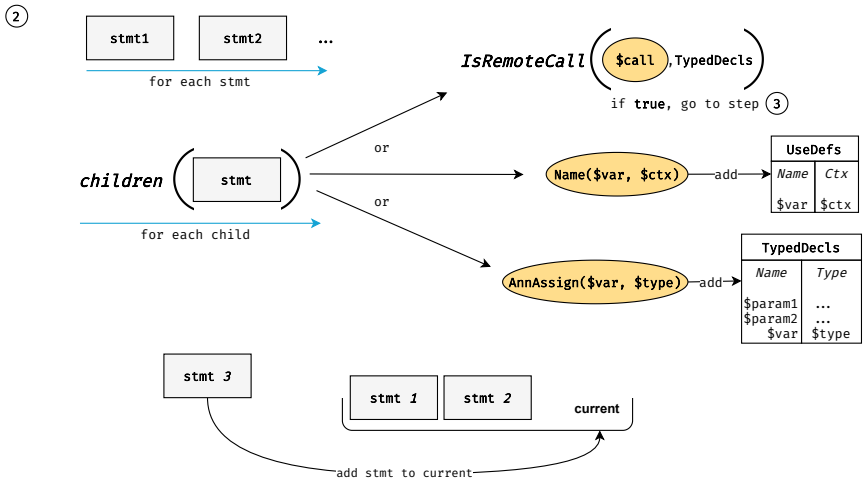


Figure 4.7: Visualization of step 2 of `SplitFunction` algorithm.

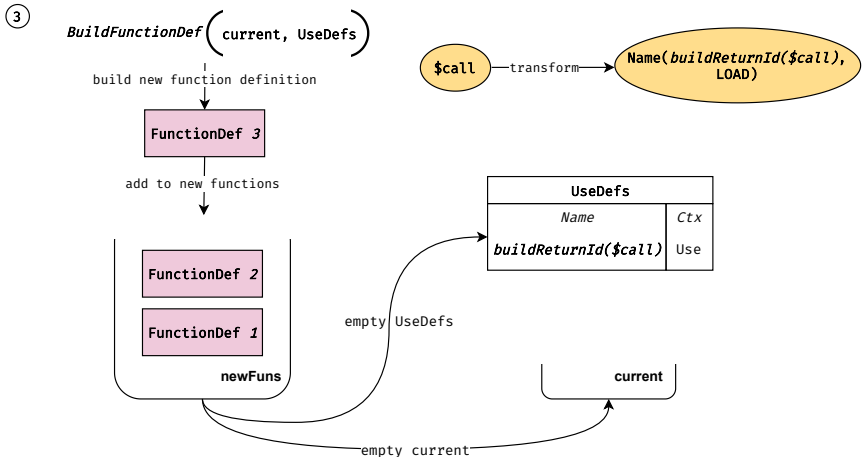


Figure 4.8: Visualization of step 3 of `SplitFunction` algorithm.

If we identify a child node as a remote call, we perform a split by building a new function definition. This function definition is created based on the current list of statements alongside its variable usages and definitions list. We empty the current list of statements as well as the list of usages and definitions. However, to the latter, we add a reference to the return result of the remote call. For the call `item.remove_stock(amount)`, the return id would be `remove_stock_return`. By adding it as the first entry in the (emptied) `UseDefs` list, we ensure that it will be the first parameter of the *next* function definition. Secondly, we transform the remote call node `$call` to a `Name` node referring to that return variable. For example, given the following statement: `x = 10 * item.remove_stock(amount)`, these operations ensure that the next function definition has the following signature and first statement:

```
1 def fun_name_x(remove_stock_return):
2     x = 10 * remove_stock_return
```

Finally, when we traversed *all* statements of the original function definition, we

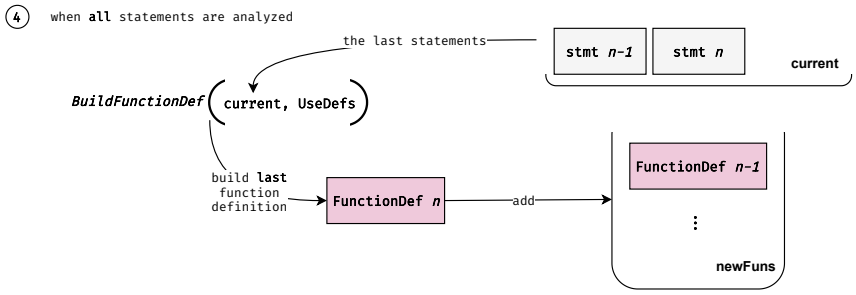


Figure 4.9: Visualization of step 4 of `SplitFunction` algorithm.

built the final function definition from the remaining statements (Figure 4.9. The output of the complete algorithm is the list of new function definitions.

To ensure coordination and continuation of these generated function definitions, StateFlow builds a state machine that encodes where to go (i.e., which operator), what to do (i.e., which function to invoke), and what to store (i.e., which local definitions are stored). Essentially, this state machine connects the new function definitions to simulate the syntax of the original definition and looks similar to the call graph shown in Figure 4.3. A function with *only* sequential control flow and n remote calls splits into $n+1$ new functions. This algorithm assumes one remote call per statement. That means call nesting and multiple remote calls per statement are not supported. However, simple desugar rules could be adopted to deal with this:

```python
def nested_call(item: Item):
    result = item.remove_stock(item.get_stock())

def nested_call_desugar(item: Item):
    x = item.get_stock()
    result = item.remove_stock(x)
```

```python
def multiple_calls(item: Item):
    result = item.get_stock() + item.get_price()

def multiple_calls_desugar(item: Item):
    x = item.get_stock()
    result = x + item.get_price()
```

In this desugared form, the proposed splitting algorithm can be applied.

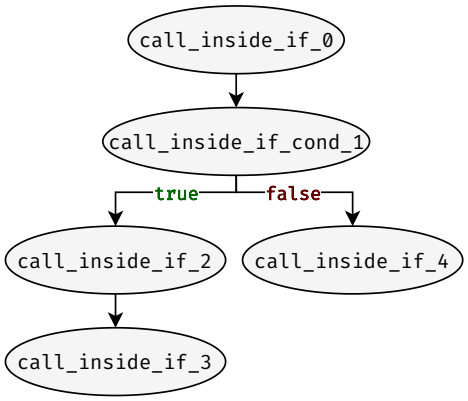
4.3.2 Conditional splits

In the previous section, we introduced the splitting algorithm. Until now, we assumed that all function definitions that are a candidate for this splitting algorithm have only sequential control flow. In this section, we introduce the support for conditional (non-looping) control flow. In other words, we show how StateFlow deals with if-statements. We consider two scenarios for an if-statement; there is a remote call *inside* the body of the if *or* there is a remote call in the *test* expression of the if. Again, we start by showing some examples before explaining the splitting approach.

Figure 4.10 shows a split based on a remote call *inside* the if-statement. The left snippet shows the original code whereas the right shows the code *after* splitting. The figure also includes the call graph for the function that was split. Starting with the original method `call_inside_if`, line 2 is reflected in the newly generated method `call_inside_if_0`. The test expression of the if-statement (line 3) is incorporated in `call_inside_if_cond_1`. The result of this function,

determines if either `call_inside_if_2` or `call_inside_if_4` is invoked. These functions reflect respectively lines 4-5 and line 7 from the original code snippet.

```
1  def call_inside_if(item: Item):
2      stock = 10
3      if stock ⩾ 0:
4          item.remove_stock(stock)
5          return True
6
7      return False
```




```
def call_inside_if_0(item: Item):
    stock = 10
    return item, stock, {"_type": "IfSplit"}

def call_inside_if_cond_1(stock):
    return stock ⩾ 0

# call_inside_if_cond_1 is True
def call_inside_if_2(stock, item):
    remove_stock_arg = stock
    return {"_type": "InvokeMethod",
            "args": [remove_stock_arg]
            .. }

def call_inside_if_3(remove_stock_return):
    remove_stock_return
    return True

# call_inside_if_cond_1 is False
def call_inside_if_4():
    return False
```

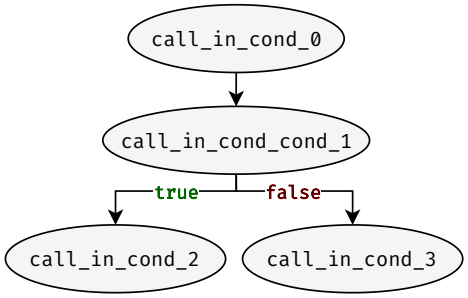
Figure 4.10: Example of splitting a function with a remote call *inside* an if-statement. Left is the original function, right is *after* splitting.

Figure 4.11 presents a split based on a remote call inside the *test* expression of the if-statement. Again, the left snippet presents the original code whereas the right snippet shows the code *after* splitting. In the original code snippet `call_in_cond`, line 2 and the call on line 3 are reflected in the new definitions `call_in_cond_0` and `call_in_cond_cond_1`. Based on the result of `call_in_cond_cond_1`, either `call_in_cond_2` or `call_in_cond_3` is executed. These function definitions reflect line 4-5 and line 6 respectively.

To support these if-statements, we adapt the presented `SplitFunction` algorithm slightly. First, we allow the `SplitFunction` algorithm to operate on a set of statements rather than a function definition. It means we skip the first step of the algorithm (Figure 4.6). Moreover, we add another trigger for splitting in step 2 (Figure 4.7). Not only do we split when we encounter a remote call, but also in the case of an if-statement. As seen in the examples, we completely remove the original if-statement and encode its behavior in new function definitions. We show this transformation in Figure 4.12. First, we create the function definition, which evaluates the **test** expression of the if-statement. This definition returns either true or false, and based on the result, a different execution path is taken. When the test expression encapsulates a remote call, like in the `call_in_cond` example, the remote call is executed before the if condition function is called. Similarly to step 3 (Figure 4.8), the return result of this remote call is passed to the if condition function. Secondly, we recursively apply the splitting algorithm to the body and the else path of the statement. As mentioned before, we build a state machine for each function that is split. This state machine encodes these conditional edges so that the correct functions are invoked in the correct order during execution.

```python
def call_in_cond(item: Item):
    stock = 10
    if item.remove_stock(stock):
        return True

    return False
```




```python
def call_in_cond_0(item: Item):
    stock = 10
    remove_stock_arg = stock
    return item, stock,
                    {"_type": "InvokeMethod",
                    "args": [remove_stock_arg]
                    .. }

def call_in_cond_cond_1(remove_stock_return):
    return remove_stock_return

# call_in_cond__cond_1 is True
def call_in_cond_2():
    return True

# call_in_cond__cond_1 is False
def call_in_cond_3():
    return False
```

Figure 4.11: Example of splitting a function with a remote call inside the *test* expression of an if-statement. Left is the original function, right is *after* splitting.

When encountering (conditional) control flow, a function is always split regardless of a remote call or not. Since the splitting algorithm is applied recursively, nested if-statements are supported.

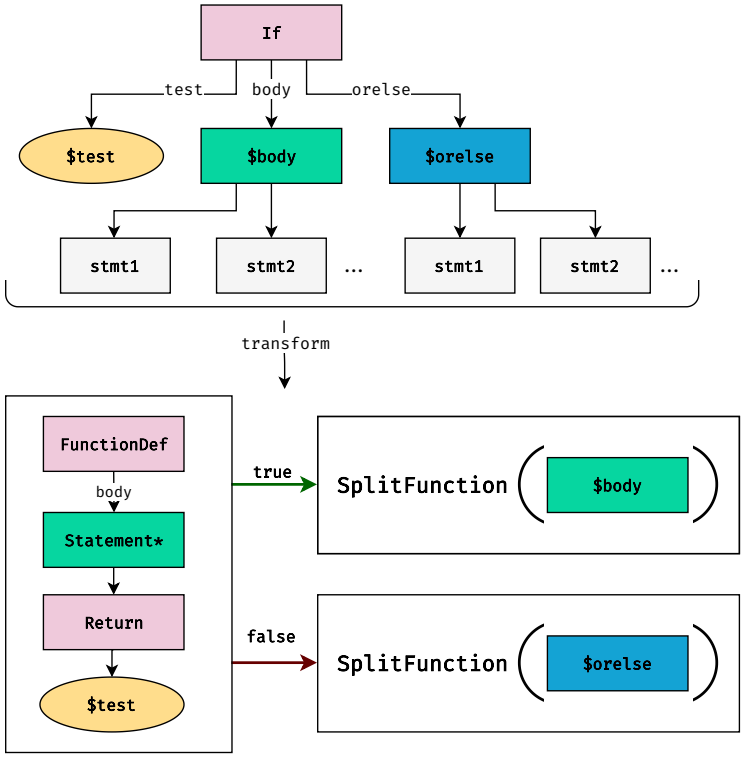


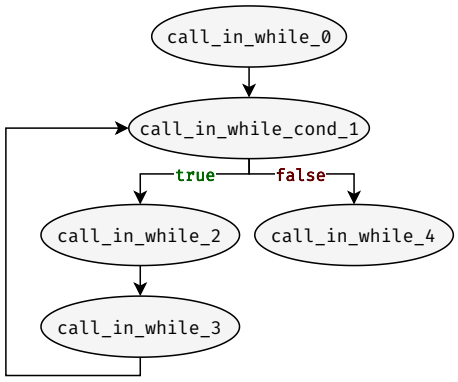
Figure 4.12: Visualization on how an `If` node is split. Most child nodes of the `FunctionDef` are omitted.

4.3.3 Loop splits

Now that we introduced the splitting algorithm for sequential and conditional control flow, we only miss support for looping control flow. In this section, we adapt the splitting algorithm to support for and while loops. Python is a dynamically typed language: types are determined at runtime. As the StateFlow splitting algorithm operates on source code, it cannot rely on this dynamic typing. Therefore, this section shows how StateFlow deals with data structures like lists by looking at its syntax. Finally, we also show how we deal with **`break`** and **`continue`** statements.

While loops The way we deal with while loops is very similar to that of if-statements. In principle, a while-loop is an if-statement where its body block is repeatedly executed until the test condition evaluates to false. Like if-statements, a while statement can have a remote call inside the test expression or inside the while body. An example 'while' split is shown in Figure 4.13. In this example, the remote call is inside the while body. The 'while' statement is com-

```
1 def call_in_while(item: Item):
2     i = 5
3     while i > 0:
4         item.decrease_stock(i)
5         i = i - 1
6
7     return True
```




```
def call_in_while_0(item: Item):
    i = 5
    return item, i, {"_type": "WhileSplit"}

def call_in_while_cond_1(i):
    return i > 0

# execute while call_in_while_cond_1 is True
def call_in_while_2(i, item):
    remove_stock_arg = i
    return {"_type": "InvokeMethod",
            "args": [remove_stock_arg]
            .. }

def call_in_while_3(i):
    remove_stock__return
    i = i - 1
    return i

# call_in_while_cond_1 is False
def call_in_while_4(remove_stock_return):
    return False
```

Figure 4.13: Example of splitting a function with a remote call inside the while statement. Left is the original function, right is *after* splitting.

pletely removed from the code (line 3), instead the test expression is evaluated in `call_in_while_cond_1`. If this function evaluates to true, we execute both `call_in_while_2` and `call_in_while_3` are before we evaluate the test expression again. Finally, when the test expression evaluates to false `call_in_while_4` is executed. Notice how we introduced an edge in the state machine introducing a cycle in executing this complete function.

To support these while loops, we adapt the splitting algorithm to execute a split when we encounter a while statement. Secondly, the while statement is removed from the abstract syntax tree and replaced with new function definitions. This

transformation is shown in Figure 4.14. First, we build a new function definition that evaluates the test expression of the while loop. This test expression can also evaluate the result of a remote call. Secondly, the body of the while statement is (recursively) split, and an edge is added to cycle back to the test expression function. Note that, a while statement also includes an else clause (the `orelse` node). This else clause is executed *after* the while loop, only if the while loop has not been terminated using a **`break`** statement. We recursively split the statements inside this `orelse` node as well. Finally, we build an edge that denotes the path once the function with the test expression evaluates to false. This edge is linked to the *next* function definition. The splitting algorithm follows the order of the statements in the code; therefore, we will add this edge after generating the next function definition.

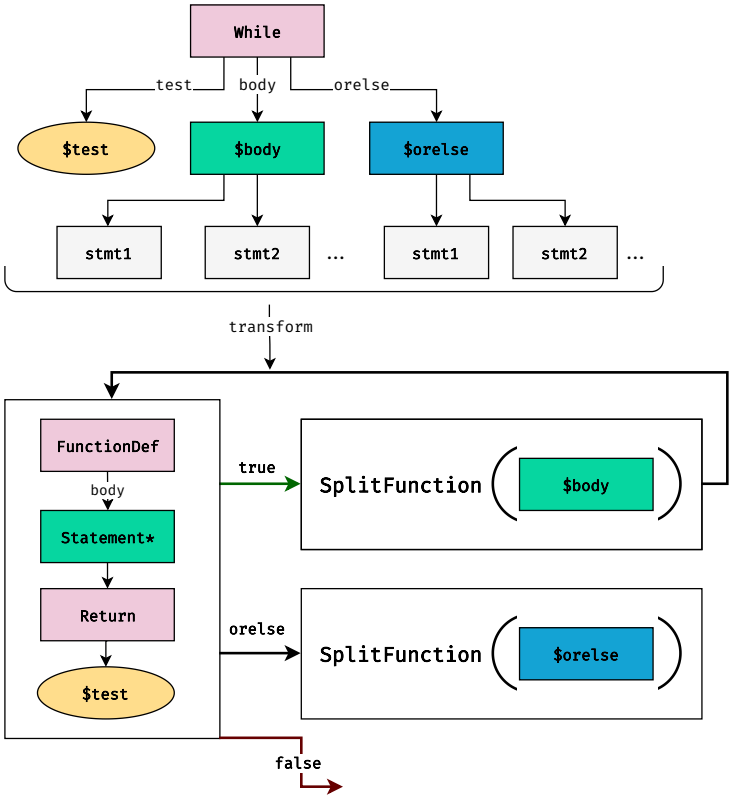


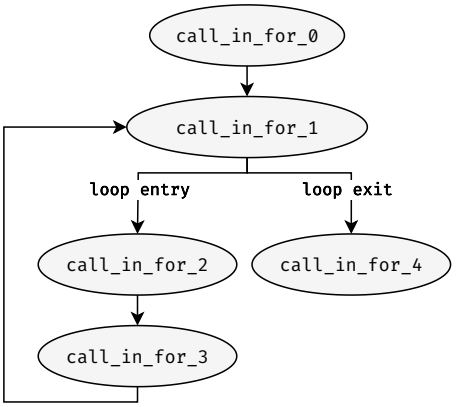
Figure 4.14: Visualization on how an `While` node is split. Most child nodes of the `FunctionDef` are omitted.

For loops Where if and while statements rely on a test expression, for loops rely on iterators. Simply put, an iterator is a data structure that allows for traversal. In a for loop, we repeatedly evaluate this iterable, and its result is stored in a target variable (i.e., **`for`** `target` **`in`** `iterable`). In each iteration, the for loop's body is executed *until* the evaluation of the iterable throws an **`StopIteration`** exception [5]. When StateFlow splits such a function, we remove the actual line (i.e., **`for`** `target` **`in`** `iterable`); however, we simulate the semantics of this construct in the newly generated function definitions and the corresponding state machine.

[5] This is how Python handles iterators and does not necessarily generalize to other programming languages.

We show an example of a 'for split' in Figure 4.15. In the first function definition of the split, `call_in_for_0`, the iterator is 'prepared' by calling the `iter(iterable)` function. `iter()` is a built-in Python function that returns an iterator for a given object. As mentioned before, this iterator can be iterated one element at a time. This is similar to what the Python interpreter executes under the hood. `call_in_for_1` is executed iteratively before the body of the for loop. In this method, the next element in the iterable is retrieved and returned. Whenever the **`StopIteration`** exception is thrown, the loop is exited. `call_in_for_2` and `call_in_for_3` represent the body of this for loop (line 4), whereas `call_in_for_5` encapsulates the statements *after* the for loop (line 5).

```
1 def call_in_for(items: List[Item]):
2     total_price = 0
3     for item in items:
4         total_price += item.get_price()
5
6     return total_price
```




```
def call_in_for_0(items: List[Item]):
    total_price = 0
    iter_1 = iter(items)
    return items, total_price, iter_1,
            {"_type": "ForLoopSplit"}

def call_in_for_1(iter_1):
    try:
        item = next(iter_1)
    except StopIteration:
        return {"_type": "StopIteration"}

    return item, iter_1

# loop entry, metadata for get_price call
def call_in_for_2(item):
    return {"_type": "InvokeMethod",
            "args": []
            .. }

def call_in_for_3(get_price_return):
    total_price += get_price_return
    return total_price

# loop exit
def call_in_for_4(total_price):
    return total_price
```

Figure 4.15: Example of splitting a function with a remote call inside the for statement. Left is the original function, right is *after* splitting.

To support these loops, initial adoptions are similar to those of if and while statements. We perform a split whenever we encounter the for construct in step 2 of the splitting algorithm (Figure 4.7). We show the transformation of the original `For` node in the AST in Figure 4.16. We append the following statements to the function definition *before* the for loop:

```
1 iter_1 = ITER
2 return DECLS, iter_1, {"_type": "ForLoopSplit"}
```

where `ITER` is the `$iter` expression (see Figure 4.16) and `DECLS` are all the declarations in that function definition. Similar to a split of a remote call, we encode

metadata in the return variables (i.e. `{"_type": "ForLoopSplit"}`). StateFlow uses this metadata at runtime to determine the next action. We do not visualize adding these statements in the transformation (Figure 4.16).

Depending on the amount of for loops in the original function definition, StateFlow generates a unique name for each `iter_i` variable. We encapsulate the evaluation of the iterable in a separate function definition, similar to how if/while statements have a different definition for the test expression. In the if/while scenario, simply the test expression is returned, whereas in the for loop scenario, we 'perform the iteration' in this function. Essentially, the following definition is generated (`ForBlock` function in Figure 4.16):

```
def for_block(iter_1):
    try:
        TARGET = next(iter_1)
    except StopIteration:
        return {"_type": "StopIteration"}

    return TARGET, iter_1
```

Where TARGET is the `$target` expression. For simplicity, we omit the corresponding AST of this code in the transformation figure 4.16. The transformation recursively splits the `body` and `orelse` clauses, and we create the proper edges. Another edge is created from the last function definition from the body to the function which performs the iteration. Finally, we create an edge for the loop exit. The loop exits when either a **`break`** or **`StopIteration`** is encountered. We connect this edge to the function definition *after* the for loop.

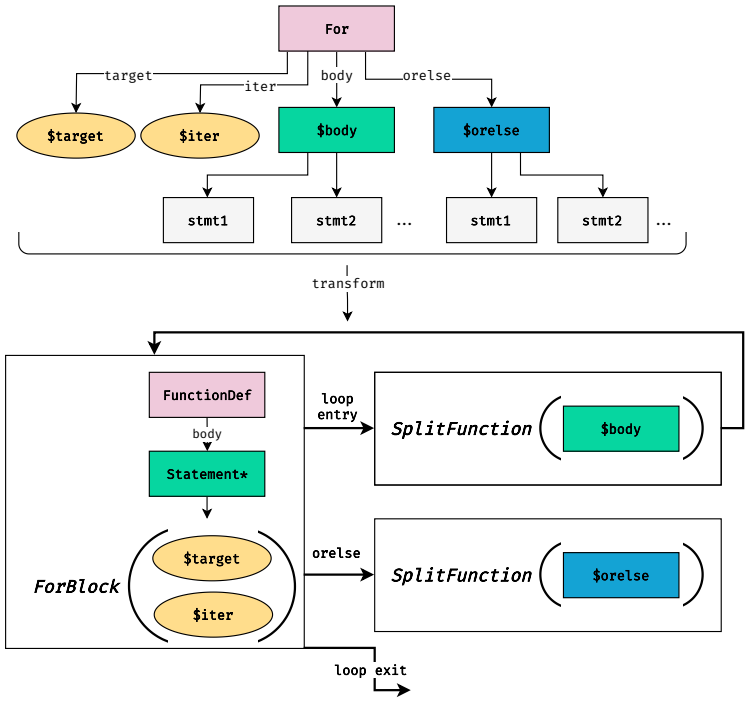


Figure 4.16: Visualization on how an `If` node is split. Most child nodes of the `FunctionDef` are omitted.

Lists Python is a dynamically typed language, and types are only determined at runtime. However, the splitting algorithm by StateFlow operates on source code; therefore, variable types need to be statically derived. A developer can generally use typed assignments to deal with container data structures (e.g., list, arrays, tuples, dictionary). In this scenario, StateFlow does not care about the right-hand side of the assignment. For example:

```
item: Item = items[0:3][0:2].pop()
```

StateFlow determines the type of `item` by looking at its type hint. It does not require any knowledge on the data structure that `item` was retrieved from. There are also scenarios where the type (hint) is not directly available. For example, the following expression (line 2):

```
1 items: ComplexContainer[Item]
2 items.get_first_element().remove_stock(10)
```

Determining the type of `items.get_first_element()` requires knowledge on both the syntax and the semantics of the underlying data structure [6]. We need to know that the `get_first_element()` method returns a single element of type `Item`. It is not trivial to derive this and might require external type checkers like MyPy [7]. Due to this complexity, StateFlow does not support these kinds of expressions out of the box. However, StateFlow does support type inference on list slices as this is a common Python expression. In the following code:

```
items: List[Item]
items[0].remove_stock(10)
```

StateFlow recognizes the subscript on the variable with a list type and therefore statically derives `items[0]` to be of type `Item`. Consequently, the function is split because of the remote call `remove_stock`. Similarly, we derive the type of the target variable in a for loop when dealing with lists. By deriving this type and storing it in `TypedDecls` (see Figure 4.7), a remote call on this variable can be recognized (see Figure 4.4).

```
# items: List[Item]
for target in items:
    # We know target is of type Item
```

In a non-list scenario, developers have to annotate the variable inside the body. For example, when iterating over a dictionary:

[6] This assumes the underlying data structure does not use types; otherwise, the type can be derived syntactically.

[7] A popular static type checker http://mypy-lang.org/.

```python
# data is type Dict[str, Item]
for k, v in data.items():
    v: Item = v
```

`v` is explicitly annotated with type `Item`. Therefore the splitting algorithm adds this variable to `TypedDecls`, which might trigger a split in subsequent statements.

Continue or break When working with loops in Python, one can use the `continue` and `break` statements inside its body. The `continue` statement exits the current body of the loop and returns to the `test` or `iter` expression. Alternatively, the `break` statement exits the current loop and continues executing the code *after* the loop. Using a `break` statement also skips the `else` clause of the loop.

StateFlow removes the while and for syntax and encodes its behavior in new function definitions. As a consequence, using `continue` and `break` inside these new function definitions results in invalid code. Therefore, these statements are also removed and replaced with return statements. A `continue` statement is replaced with `return {'_type': 'Continue'}` and `break` with `return {'_type': 'Break'}`. Whenever StateFlow encounters such a return statement, either the test/iter function is executed or the first function *after* the while/for loop for dealing with respectively `continue` and `break`.

4.3.4 State requests

So far, we have only seen interaction with remote entities in the form of calls. However, in Python, one can also access or update attributes of another object. For example:

```python
# item of type Item
item.price # Attribute access
item.stock = 10 # Attribute update
```

We support attribute access by syntactically inferring these expressions in the statements of the original function definitions. Once inferred, we append some metadata to the newly generated function definitions. This metadata is not reflected in the code of corresponding definition. During execution, before this function is invoked, the accessed attributes of this remote entity are retrieved. StateFlow passes around a lightweight data structure for remote entities that stores these retrieved attribute values. This way, the original syntax in the form `obj.attribute` does not need to be transformed. We present more details on execution and internal representations in Chapter 5.

In StateFlow we do not support attribute updates of remote entities; however, adding this is straightforward. One can set an attribute via an explicit method call `obj.__setattr__(key, value)`. Therefore, attribute updates can be desugared to this explicit call form and then we treat it as a remote call.

4.3.5 State machine

The splitting algorithm transforms a single function definition into multiple. A state machine encodes how these new function definitions interact with each other. This state machine can be seen as a call graph, deciding on the order of function evaluation. Some examples of this state machine have been given in Figure 4.10, 4.11, 4.13 and 4.15. Essentially, this state machine simulates the semantics of the original (i.e., before splitting) function definition.

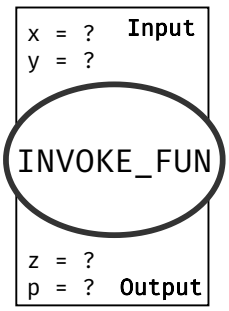
In this state machine, the states represent the different function definitions. There are different types of transitions between states (i.e., edges). The table on the right shows all these transition types. The most basic transition is `next`, and regardless of the outcome the current state moves to the next state. This transition mimics sequential control flow in the original function definitions. The `true` and `false` edges encapsulate conditional transitions. Depending on the binary outcome of the test expression function, either the `true` or `false` transition is taken. Similarly, `loop entry` and `loop exit` transitions encapsulate respectively executing the body of the loop or exiting the loop. The `orelse` transition represents the `else` clause of the if, for and while statements. Finally, the `remote call` transition encodes that in between the two states it connects, a remote call to another stateful entity is executed.

Edge type
`next`
`true`
`false`
`loop entry`
`loop exit`
`orelse`
`remote call`

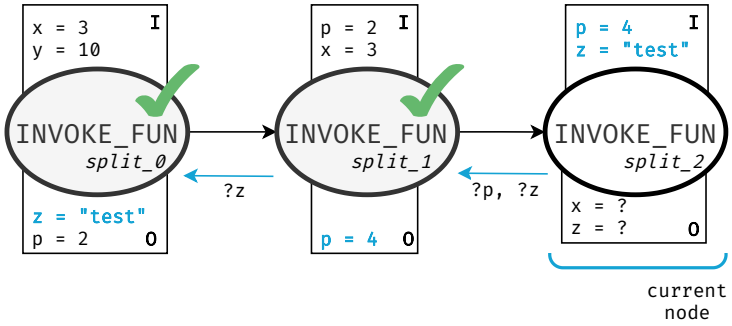
4.3.6 Execution graph

The state machine encodes how the split functions transition such that we simulate the original function definition. We extend the state machine with specific instructions for execution and label this the execution graph. This graph stores local variable definitions, return variables, and metadata on the invocations. This graph instructs the execution model to invoke the correct functions at runtime (see Chapter 5). Note that functions that are not split, do **not** have an execution graph as their execution is straightforward. This section elaborates on all the different node types and how StateFlow traverses this graph.

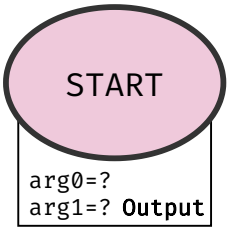
Invocation nodes The most common node in this graph is a `INVOKE_FUN`. This node instructs the execution model to invoke a function and has input and output variables. The node's input resembles the arguments of the function it ought to invoke. The output variables are the declarations of that function **and** potentially metadata (i.e. `{"_type": "ForLoopSplit}`. These are returned upon function exit as shown in Figure 4.5. The metadata variable is omitted from the figure on the right. After function invocation, the output variables are stored inside the node, and the node is marked as finished. The next node in the graph is selected based on the metadata in the function output variables. The traversal of this graph is dynamic since this metadata is returned during runtime and might be different for each invocation. Upon reaching a `INVOKE_FUN` node, we retrieve input variables from previously visited nodes. We traverse the graph backward, and match the names of the latest output variable to the input variable names of



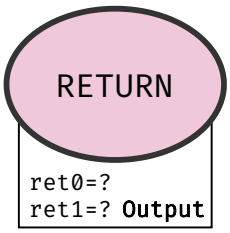
the current node. These output variables correspond to the latest declaration of that variable. The figure below shows an example traversal.

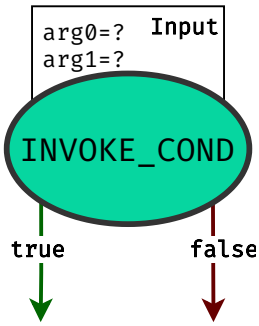


It shows the function `split` which has been split into three definitions. After invoking `split_0` and `split_1`, `split_2` needs to be invoked. This function has the variables `p` and `z` as input. The execution graph is traversed backward to find the latest versions of output variables with the same name. Afterward, once the function is invoked, the output variables are set, the node is marked as finished, and the next node in the graph is selected. We select the next node by traversing the outgoing edge of the current node. This backward traversal approach is somewhat naive and impacts the performance for large execution graphs.The evaluation Chapter 6 discusses the impact of this approach.



Start and return nodes The first node of each execution graph is always a `START` node. This node does not invoke or execute a function and only stores the initial function arguments as output variables. These function arguments match the parameters defined in the original (i.e., before splitting) function definition. By storing it in the output of this start node, child nodes can access it by traversing the graph backward. Execution of the graph always ends whenever we reach a `RETURN` node. As the original function definition might have multiple returns, so does the execution graph. This node stores the return variables, which are passed back to the client.



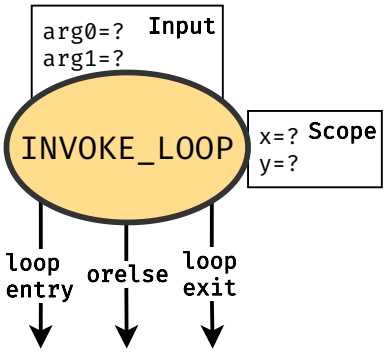


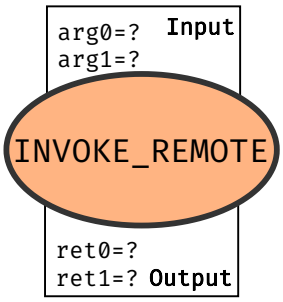
Conditional nodes The `INVOKE_COND` node encodes the function definition storing the test expression for an if statement. Similar to invocation nodes, it evaluates the function based on input variables. However, instead of having multiple return variables, this function is supposed to return either true or false. The execution graph traverses the corresponding edge and selects the next node. Only the previously executed nodes are visited whenever a child node needs to find the correct input variable. Therefore an execution graph might have unvisited subgraphs.

Loop nodes The `INVOKE_LOOP` node encodes the function definition which is the 'starting point' of a loop. For a while statement this is the function with the test expression, whereas for a for statement this is the function traversing the iterator.

The correct edge is traversed based on the output of the function. Different from the nodes seen so far, this node is revisited and different output edges might be traversed. For example, the loop entry path is traversed x times before the loop is exited.

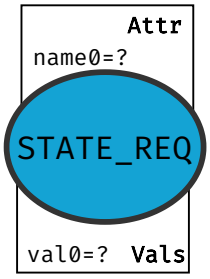
As loop nodes introduces cycles in the execution graph, backwards traversal becomes less trivial as each iteration might override variables of the previous execution path. Therefore the `INVOKE_LOOP` stores a list of all latest output variables defined in the body of the loop (i.e. the nodes in the subgraph of `loop entry`). In essence this list resembles the scope of this loop. Whenever a subsequent node outside of the loop requires a variable defined *inside* the loop it can be found in this 'scope' list. Therefore all nodes inside the loop body do not need to be traversed.





Remote call nodes We encode a call to a remote stateful entity in the `INVOKE_REMOTE` node. This node is similar to the `INVOKE_FUN` node. The only difference is that a function of a remote entity is invoked. In practice, this is the moment we exit the current operator and move to the operator storing the 'remote' entity. After invoking the function on this entity, we move back to the original operator to continue traversing the execution graph. Properties like the key of the remote entity and its function arguments are dynamically determined by the node executed before `INVOKE_REMOTE`. This metadata is encoded in a dictionary structure as shown in Figure 4.5.

State requests The final node type is the `STATE_REQUEST`. This node requests an attribute of a remote entity. Similar to `INVOKE_REMOTE`, it triggers a switch between dataflow operators to request the attributes. This node does not have output variables; instead, it sets the retrieved values for the requested attributes in the reference object of the remote entity. This way, attribute access like `item.price` is made possible.



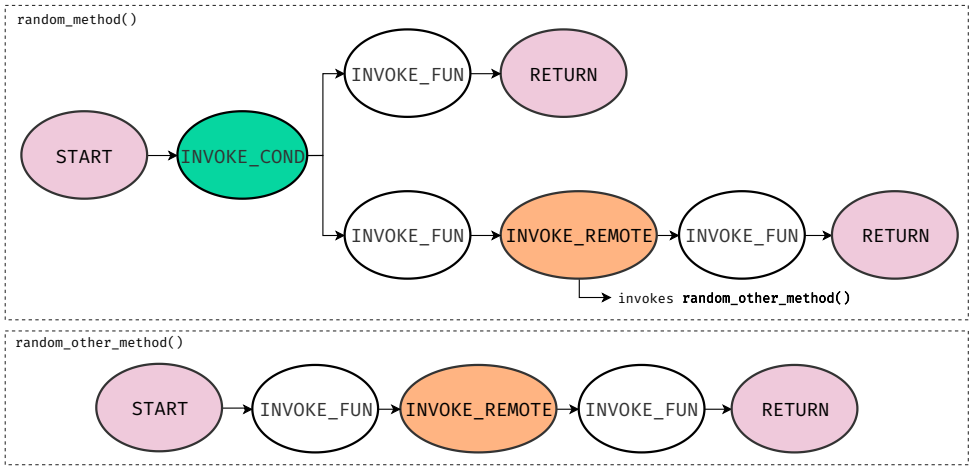
Graph visualization We hide the execution of the splitting algorithm and the creation of the state machine and execution graph for the developer. StateFlow offers utility methods to generate visualizations of the state machine and execution graph to give the developer insights and provide transparency. We print these visualizations in the well-known DOT format [8]. The DOT format can be visualized in many (online) tools. These utility methods can be used during execution to debug function invocations.

[8]The DOT format is graph description language https://en.wikipedia.org/wiki/DOT_(graph_description_language)

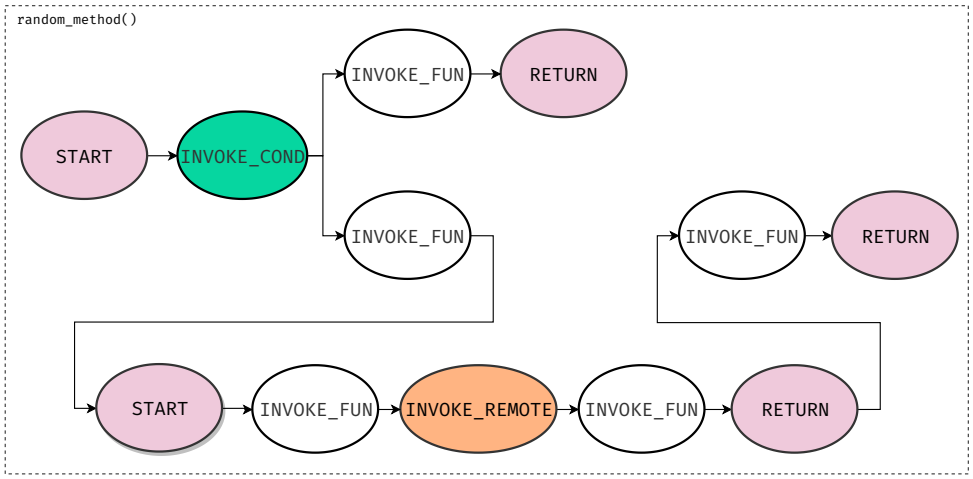
4.3.7 Nested split functions

An execution graph executes all the functions for a function declaration that has been split. Until now, we assumed that remote calls in such execution graphs are **not** split. To support nested 'split' functions, we merge execution graphs. By merging the execution graphs, execution can be done similarly to a single execution graph.

After generating the execution graphs of all split functions, we iterate over all remote call nodes and identify if the function that this node is supposed to invoke is also split. If that is the case, we retrieve the execution graph of this remote call and merge the two. For example, consider the following two execution graphs which we need to merge as we also split the remote call `random_other_method()` inside the method `random_method()`.



We merge these two graphs by removing the `INVOKE_REMOTE` node and connect its incoming edge to the `START` node of the other execution graph. All `RETURN` nodes of the other execution graph are connected with the node *after* the removed `INVOKE_REMOTE` node. The figure below shows this merge:



As each execution graph simulates the scope of a function, these merged execution graphs should not have access to each other scope. Therefore, each execution graph gets a unique id. Whenever a node requires a particular input variable, it

can safely traverse the merged execution graph and skip the nodes with a different id. `RETURN` nodes are exempted from this as they store return variables rather than variables in the scope of a function.

4.3.8 Splitting applied to the running example

In Section 3.5 we show the example of two stateful entities `User` and `Item`. In these entities, only the method `def add_to_basket` is a candidate for the splitting algorithm. All other methods can be invoked without requiring a remote call. The state machine and execution graph for this function *after* splitting is show in Figure 4.17.

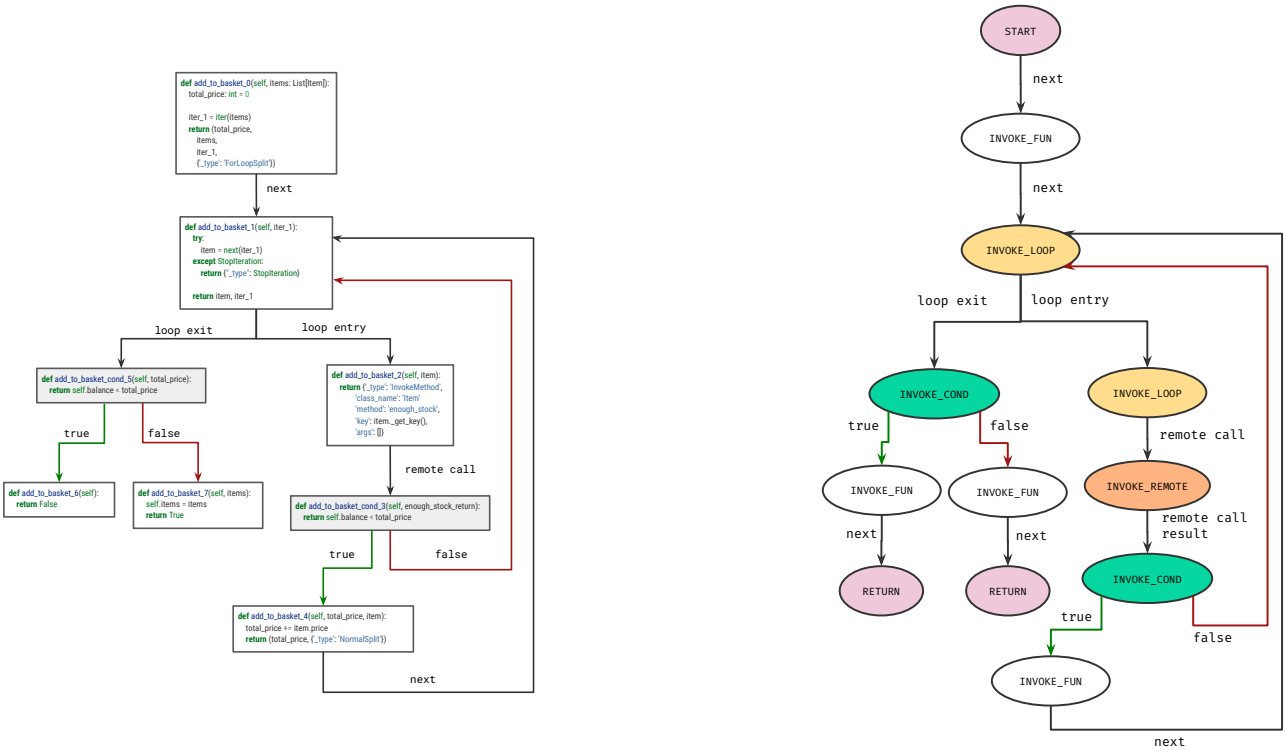


Figure 4.17: The state machine (left) and execution graph (right) after the `def add_to_basket` method has been split in the `User` stateful entity.

4.4 Intermediate Representation

The StateFlow compiler pipeline results in an intermediate representation for the defined and annotated classes. This IR captures all Python classes and translates to a **stateful dataflow graph**. In summary, to get to this point, developers have to write the following code:

```python
from stateflow import stateflow, Dataflow

@stateflow
class User:
    ...

@stateflow
class Item:
    ...

flow: Dataflow = stateflow.init()
```

Developers write functionality in the form of (annotated) classes. These classes are recompiled into a stateful dataflow graph with `stateflow.init()`. The `Dataflow` object stores (re)compiled classes, execution graphs, and metadata for each annotated Python class. Finally, this IR is the starting point to either build the client-side of the application **or** to pick a target runtime system to which we compile the IR. We extensively discuss both options Chapter 5.

4.5 Limitations

Although the compiler pipeline of StateFlow opens up a set of possibilities, it also imposes some limitations for the developer using it. In this section, we identify several of these limitations and propose workarounds if available.

Python constructs The splitting algorithm of StateFlow does not cover all Python constructs. For example, we do not handle a try/catch construct. StateFlow covers 72% of all statement constructs, and 74% of the expression constructs [9] in its transformation and analysis phases. Moreover, some nested constructs, like remote call nesting or multiple remote calls in a single statement, are not supported. Covering all constructs is mostly an implementation effort, and the current StateFlow codebase already spans around 10000 lines of code. We believe that most of these uncovered constructs can easily be implemented using the ideas presented in this chapter. For example, a try/catch construct can easily be split into multiple function definitions and have a distinct node type in the execution graph. Alternatively, desugar rules can be applied to transform the code into a form that is supported for the splitting algorithm. Finally, we argue that the constructs currently supported cover the most important programming principles.

Serializability StateFlow requires all instance attributes to be serializable. Whenever we split a function, its local declarations also need to be serializable. For example, storing a database connection or local pipe inside a variable is not supported. In general, not complying with this results in unexpected behavior or runtime errors. In distributed programming, serialization is often a hard requirement as code or data is sent around the network.

[9] Based on the official Python grammar https://docs.python.org/3/library/ast.html

Remote entity instantiation In StateFlow, we always assume that references to remote entities are passed around as variables and not instantiated inside a class method. Consequently, the developer always has to instantiate an entity on the client-side before it can be passed as an argument to a function. In Python, an object instantiation can be made explicit by a method call. For example, `Item.__init__(item, init_args)` initializes the variable `item`. StateFlow may support object initialization by identifying it, desugaring it into its explicit form, and treating it as a call to a remote entity.

Immutablity Whenever we split a function, we recommend treating state (i.e., instance attributes) as immutable. For example, consider the following stateful entity and its state:

```
@dataclass
class Room:
    reserved: bool = False
    price: int  = 10

@stateflow
class Hotel:

    def __init__(self):
        self.room: Room = Room()

    def reserve(self, user: User):
        the_room: Room = self.room
        user.pay(the_room.price)
        the_room.reserved = True
```

Notice that `Room` is **not** a stateful entity and is simply a container class storing some data. As the reserve method has a remote call, it is split and results in the following two methods:

```
def reserve_0(self, user: User):
    the_room: Room = self.room
    user.pay(the_room.price)
    return the_room, {"_type" : "InvokeMethod",  ...}

def reserve_1(self, the_room: Room, user_pay_return):
    user_pay_return
    the_room.reserved = True
```

Due to serialization, the 'link' between `the_room` and `self.room` is broken. In `reserve_0`, these variables share the same memory address. However, after that method is exited and both variables are separately serialized their shared memory address is gone. Updates to `the_room` are not reflected in `self.room` and the other way around. Treating state as immutable and explicitly updating it prevents this issue. In this example, appending the statement `self.room = the_room`

in `reserve_1` resolves the issue. If `Room` is an remote entity, this would not be a problem as updates to its state are explicit.

Dynamic code Python code is rather dynamic. For example, one can add a method or attribute *after* the class definition. In general, Python allows for easy manipulation of class or function definitions at runtime. StateFlow operates on (static) code, and therefore these kinds of dynamic operations are not supported.

Polymorphism StateFlow does not support polymorphism. It assumes that classes do not inherit functionality from a parent class. To support this, StateFlow needs to extend its analysis and function splitting algorithm to identify parent classes and (recursively) split their functions if necessary.

Chapter 5

Executing Stateful Dataflow Graphs

In Chapter 4, we introduced a compilation pipeline to transform object-oriented Python code to a stateful dataflow graph. In this chapter, we propose methods to execute these dataflow graphs in line with the introduction given in Figure 3.2. First, we explain how StateFlow translates its intermediate representation to different runtime systems in Section 5.1 and 5.2. In Section 5.3, we explain how StateFlow utilizes the same IR to provide a client-side interface. This interface allows easy interaction with the deployed application. We show how developers can unit-test their classes in a local execution environment in Section 5.4. Section 5.5 highlights the tools and configuration files StateFlow offers to ease the deployment of end-to-end cloud applications. We end this chapter with a summary of all implemented client interfaces and runtime systems in Section 5.6.

5.1 Execution in StateFlow

StateFlow follows an event-driven approach for execution: events traverse the stateful dataflow graph and trigger method execution. To port and execute the stateful dataflow graph on top of the runtime system, we propose a set of *building blocks*. These building blocks allow loose coupling with the underlying runtime systems. StateFlow encapsulates most of the runtime functionality in these building blocks, including event execution, serialization, and routing. It has several advantages to 'lift' functionality to StateFlow. First, integrations with runtimes require little code, and adding new runtimes is trivial. Second, we can add new functionality to these building blocks, such as monitoring and logging, without changing any integration code. Finally, as StateFlow handles the actual event execution, runtime behavior is consistent across all runtimes.

Execution in StateFlow operates at the *event level*: building blocks solely perform transformations on events and possibly state. However, StateFlow does not take charge of runtime features such as (auto)scaling, event streaming, fault-tolerance, consistency and state management. We rely on the underlying runtime for these features. Each runtime offers different guarantees and features with certain trade-offs.

As mentioned in Section 3.2, the stateful dataflow graph encapsulates the defined Python classes in operators (i.e., nodes or vertices in a graph). These operators do not perform any computation without being triggered by an incoming event. In OOP terms, an object does not perform computation before its method is

explicitly called. We expect these events to be sent from a client via event streaming platforms such as Apache Kafka. Below, we summarize the general procedure for events entering the stateful dataflow graph. Note, this is the procedure for invoking stateful entities: the most common type of event. For each incoming event:

1. Route event to the correct operator.
2. Retrieve the state for the given key.
3. Construct the stateful entity and execute the method.
4. Store the (updated) state.
5. For the resulting event, route to the next stateful entity or send back to the client.

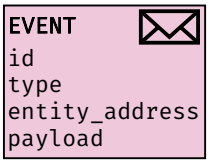
Essentially, for each stateful dataflow graph, we *dynamically* build an application encapsulating these steps on the runtimes. For example, in a streaming system like Apache Flink, we implement step 1 and 2 using `map()` and `keyBy()` operations. Then for each stateful entity in the IR, we build a separate Flink `process()` operator to cover step 3 and 4. Finally, a `map()` operation covers step 5.

Depending on the runtime, steps are skipped or combined. For example, in a FaaS solution like AWS Lambda, we do not necessarily have the notion of 'operators', and therefore we skip step 1 and query a database directly. The rest of this section elaborates on the constructs and building blocks in StateFlow.

5.1.1 Constructs

Before we dive into the building blocks of StateFlow, we elaborate on some of the constructs and techniques it uses. Most of these constructs are common in the world of distributed systems and programming.

Events Events are the core of event-driven systems such as dataflow systems. In StateFlow, we use events primarily to invoke methods of stateful entities. In short, whenever a client invokes the method of a stateful entity, we generate an event encoding this request. This event is sent to a target (runtime) system to perform the actual invocation of that method. The return values of a method invocation are sent back to the client in the form of an event. An event has a set of properties, as shown in the figure on the right. An event *id* is a unique identifier and allows matching event requests to its responses. The *entity address* encapsulates the 'virtual location' of a stateful entity. We discuss this in more detail in the next paragraph. The event *type* describe the purpose of each event, for example, 'invoke a method'.



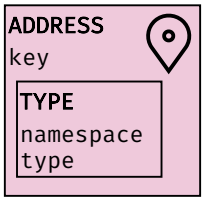
Besides method invocation, an event might have a different purpose, such as creating a new entity. We present a complete summary of all event types in Table 5.1. Most of the types are self-explanatory. We use the `InvokeMethodSplit` type to indicate the invocation of a method that StateFlow has split. Finally, the event encodes a payload which encodes all additional (meta)data. Most importantly, it encodes the call arguments and return variables when invoking a method and it stores the execution graph of a function that is split.

Table 5.1: All types of events.

Type	
Request	**Reply**
`InitEntity`	`SuccesfulCreateEntity`
`InvokeMethod`	`SuccesfulInvocation`
`FindEntity`	`FailedInvocation`
`GetState`	`FoundEntity`
`SetState`	`EntityNotFound`
`InvokeMethodSplit`	`SuccessfulStateRequest`
`Ping`	`Pong`

The invocation of a non-split function is relatively simple: an event enters the runtime system, performs some computation in an operator, and then leaves the system again. However, for a split function, we need to invoke multiple functions at different operators. All this information is encoded in the execution graph and embedded in the corresponding event. As such an event flows through the runtime system, StateFlow traverses the enclosed execution graph, determining *where* to go and *what* function to invoke.

Entity address To route events to the correct entity, we use the concept of an entity address. This concept is inspired by the function address used in Flink Statefun [1]. In a local programming context, we keep references to objects (i.e., stateful entities) in the form of variables. In a distributed setting, the runtime location of an entity, or its state, is often not exposed to the user. Instead, we maintain a reference in the form of an entity address. In this address, we store the (static) type of the entity. This type corresponds to the class name and its namespace. Currently, all class types are in the same namespace `globals`. However, we envision that developers should be able to set this namespace via the class annotation (i.e. `@stateflow(namespace="webshop")`. A namespace allows duplication of class names. The address also encodes the key of the stateful entity. We derive an entity key by calling its `def __key__()` method. Using this 'entity address' scopes the corresponding events to that specific entity. The concept of virtually addressable entities is not new and has been proposed before by Hellerstein et al. [2019b].



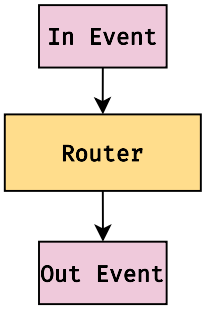
State In StateFlow we define the state of an entity as a set of keys/values corresponding to the attributes of that entity. In the object-oriented context, this corresponds to the instance attributes of an object. We operate on this state through method invocations. In other words, given a method invocation event and the state of an entity, we can compute the result of this method. We can do this computation *anywhere* as long as the corresponding class definition is available. This approach simulates invoking a method on an object in a local context. Depending on the runtime system, the state is stored in a database or the stateful operators of a dataflow system.

[1] https://ci.apache.org/projects/flink/flink-statefun-docs-release-2.2/concepts/logical.html#function-address

Serialization As we are dealing with a distributed setting, we need to send and receive events over the network. Network communication requires serialization and deserialization techniques. In StateFlow, we currently support `JSON`, `pickle` and `Protobuf` serialization. Moreover, we allow developers to add other serialization frameworks. For that, one only has to implement a serialization and deserialization method of the `Event` type.

5.1.2 Building blocks

Now that we established a set of constructs used in StateFlow, we introduce building blocks. These building blocks allow easy integration with different target runtime systems. As an advantage, we can add or modify the functionality of these building blocks without touching any of the code related to the runtime integration. StateFlow offers two main building blocks: *routers* and *operators*.



Routers We propose two types of routers: an ingress router and an egress router. These routers analyze the (incoming) events and output information on the route they should take. The *ingress* router processes incoming events, whereas the egress processes the outgoing events. The routers also (de)serializes the events as we assume that events enter and leave the runtime system in a serialized form. In a dataflow graph context, the ingress router selects which edge the event has to traverse or which (logical) partition it belongs to. The router uses both the event type and the enclosed entity address to determine this. For example, if we want to invoke a method on the entity `Item` with `key=jeans`, then the ingress router instructs us to traverse the edge towards the `Item` (dataflow) operator. In addition, it forwards the event to the the logical partition which stores the key `jeans`. The *egress* router determines if we need to send the return event to the client or we flow the event back into the system. The latter might be necessary when invoking a split function. The routers generalize its output such that we can also use this route information in a non-dataflow system.

Note that the routers do *not* modify the incoming event. Moreover, they are completely *stateless* which means that it keeps no persistent state; only the incoming event is required to execute the routing operation. We can easily parallelize stateless operators by distributing the incoming events. Finally, routers are completely static regardless of the intermediate representation.

For non-split functions, routers primarily decide on the route based on the *entity address* field since it only targets one entity. However, in a split function, multiple entities are involved. In that scenario, routers consider the current node in the execution graph and infer which entity it concerns. Similarly, the egress router analyzes the execution graph to decide when to go back to the client (i.e., the current node has a `Return` type) or not.

Operators The compiler pipeline, discussed in Chapter 4, results in (dataflow) operator for each annotated Python class. This operator stores the compiled class, a class description, and, if necessary, corresponding execution graphs for each

method. We extend this operator to be able to execute events. We show the general idea behind execution in Figure 5.1.

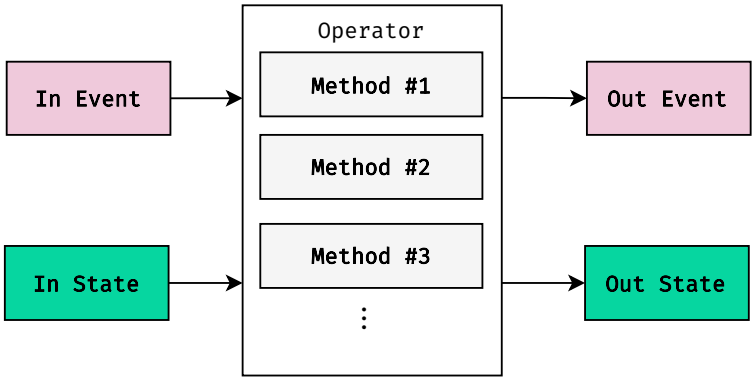


Figure 5.1: Visualization of the operator building block.

A StateFlow operator executes a class method using the current state and an incoming event. It uses the state to reconstruct the stateful entity (i.e., object), and the event encodes the correct method and call arguments. The operator outputs the updated state and the outgoing event. In this model, storage of state and computation are decoupled. The underlying runtime manages the state on behalf of the entity. This way, execution is stateless, and we can perform the computation *anywhere* as long as the event and the state are available. Industry solutions like Flink Statefun and CloudState also adopt this model [2].

StateFlow assumes all incoming events are scoped to the correct operator (i.e., the `User` or `Item` operator). The ingress router plays a role in scoping events. Moreover, we assume that the incoming state belongs to the specified entity. In other words, if we have an event with an entity address pointing towards an `Item` with the key `jeans`, then we retrieved the state of that entity. How we retrieve this state depends on the underlying runtime. In the case of a dataflow system, we assume this state is available in the scope of the operator, whereas in FaaS context, we might need to retrieve it from an external database.

Figure 3 shows how StateFlow invokes a method using the incoming event and the state. Given the input event and state, StateFlow constructs the entity and invokes the method. In object-oriented terms, we create an empty object, set the attributes, and then invoke the event's specified method. Notice that we do not initialize the object; otherwise, the `def __init__(..)` method is called, and we do not want that. Based on the output of the method, we build an outgoing event. For example, if the method throws an error, we return `FailedInvocation`. Finally, we extract the updated state from the created object. In this model, the lifetime of an entity spans *only* its method invocation.

Whenever StateFlow splits a function, it also builds an execution graph in its compiler pipeline. We copy the execution graph into the event's payload whenever a user invokes such a split function. Rather than invoking a single method, we traverse the execution graph and execute the corresponding methods. If the next node in the graph instructs to invoke a method part of another stateful entity, we leave the current operator. On the other hand, if the next node in the graph

[2]Flink Statefun: https://flink.apache.org/stateful-functions.html and CloudState: https://cloudstate.io/

```
try:
    # Construct the entity.
    constructed_entity = self.cls.__new__(self.cls)

    # Set state of entity.
    for k in self.class_desc.state_desc.get_keys():
        setattr(constructed_entity, k, state[k])

    # Call the method.
    method_to_call = getattr(instance, method_name)
    method_result = method_to_call(**arguments.get())

    # Get updated state.
    updated_state = {}
    for k in self.class_desc.state_desc.get_keys():
        updated_state[k] = getattr(instance, k)

    # Return the results.
    return InvocationResult(
        State(updated_state), method_result
    )
except Exception as e:
    return FailedInvocation(f"Exception during invocation: {e}.")
```

Listing 3: Code used to invoke a method given an event and the state of an entity.

instructs to invoke a method from the same stateful entity, we reuse the object constructed for the first invocation.

Examples In Figure 5.2, we show how the building blocks are supposed to work together to manage the execution of a non-split function. The incoming event,

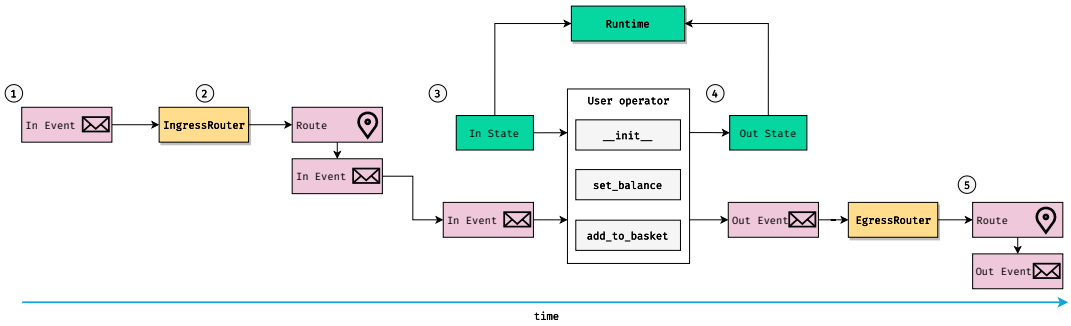


Figure 5.2: Execution of an event for a non-split function invocation. This figure shows how the building blocks work together.

which encodes the involved entity, method to invoke and call arguments, enters the runtime system and is first forwarded to the ingress router (1). The ingress router derives the 'route' of the event and encapsulates which exact stateful entity is involved (2). Depending on the runtime, this allows deciding 'where to go' or 'what to request from a database'. For example, this route enables moving the event to the partition, storing the stateful entity in a dataflow system. Given the

route, the correct state is retrieved from the runtime, and in combination with the incoming event, the operator building block is invoked ③. The operator block executes the correct method and returns the updated state plus the outgoing event ④. We also pass the state back to the runtime to store it there. Finally, the egress router analyzes the outgoing event and determines a new route ⑤. This route encodes if the event is sent back to the client or passed to the next operator.

Similarly, we show in Figure 5.3 how building blocks work together to execute a split function. The event triggering such an invocation encodes the execution graph, which is traversed as the event flows through the system. First, the event

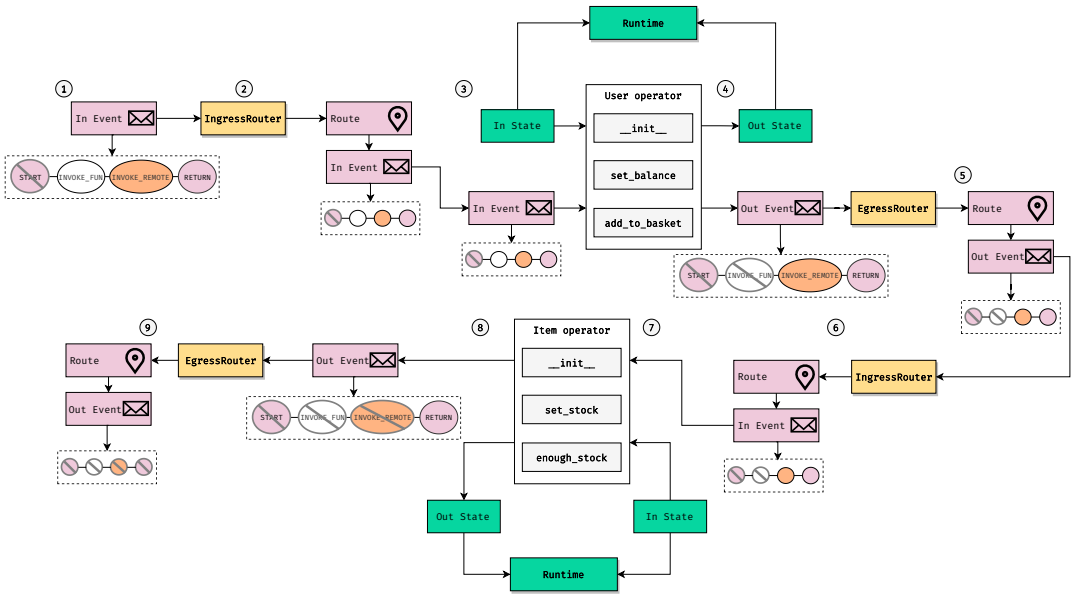


Figure 5.3: Execution of an event for a split function invocation. This figure shows how the building blocks work together.

with the execution graph enters the system ①. The `START` node in this graph is traversed on the client-side already and stores call arguments of the split function. In the ingress router block, StateFlow identifies the route of the event ②. It checks the current node in the execution graph and extracts the involved stateful entity. StateFlow retrieves the corresponding state from the runtime and, together with the incoming event, executes the operator building block ③. The operator returns the resulting event and updates the resulting state ④. Besides executing the correct code, the operator also traverses the execution graph. The egress router identifies that the execution graph has not been traversed entirely and forwards the event back into the system to the ingress router ⑤. Again, the ingress router identifies the current stateful entity in the execution graph ⑥. Like before, StateFlow passes the event and state to the operator ⑦, resulting in an outgoing event and an updated state ⑧. The outgoing event is sent to an egress router, which identifies that the current node is a `RETURN` and sends the event back to the client ⑨.

5.2 Runtimes

In this section, we elaborate on the different runtime systems StateFlow supports. We use the previously defined building blocks and constructs to propose

architectures for several runtime systems. StateFlow generates runtime implementations dynamically given the compiled stateful dataflow graph. Due to these existing blocks, the code footprints of these integrations are tiny, and supporting new runtimes is a trivial task. We make a distinction between two types of systems: dataflow systems (Subsection 5.2.1) and (Stateful) Function-as-a-Service (Subsection 5.2.2).

From a developer's perspective, switching between runtimes only requires importing a different runtime integration. Find an example in Figure 5.4:

```
from running_example import stateflow, User, Item
from stateflow.runtime import FlinkRuntime, BeamRuntime
# Initialize stateflow
flow: Dataflow = stateflow.init()

# Pick a runtime.
runtime: FlinkRuntime = FlinkRuntime(flow)
# OR, switch to Beam
runtime: BeamRuntime = BeamRuntime(flow)

# Start the runtime.
runtime.run()
```

Figure 5.4: Selecting and switching between a runtime in StateFlow.

This code generates an application on top of the selected runtime using StateFlow's IR. Most runtimes run out-of-the-box in this local environment. To deploy the runtime application onto a cluster requires more effort and differs for all runtimes. StateFlow offers guides and default configurations to ease this process.

5.2.1 Dataflow systems

In this section, we introduce the integration of StateFlow with two popular streaming processing frameworks: Apache Flink and Apache Beam. In these frameworks, developers implement their applications in the form of a streaming dataflow graph. As StateFlow's intermediate representation uses the form of a stateful dataflow graph, there exists an simple translation to this streaming dataflow graph.

For the integrations with Flink and Beam, we use their Python API. Both frameworks are not implemented in Python, and their Python API is merely a binding to their native JVM implementation. Unfortunately, this Python support is relatively new and has implications for its performance. We elaborate on these performance issues in the evaluation. Alternatively, we present an integration directly implemented using Flink on the JVM where the actual Python code executes in (remote) stateless Python functions. We opt for AWS Lambda for the latter.

Flink and Beam in Python Streaming systems operate on unbounded datastreams by means of a directed acyclic graph (DAG). These streaming dataflow graphs start with sources and end with sinks. This graph represents (user-defined) operators and performs a computation or transformation over the datastream.

Edges in this graph represent how data in the datastream flow from one operator to another. StateFlow supports calling methods from other stateful entities: this requires cycles in the stateful dataflow graph. However, streaming dataflow graphs do not support cycles. We rely on Apache Kafka, an event streaming platform, to cycle events back into the streaming graph. Thus, the combination of the streaming dataflow graph plus Apache Kafka mimics StateFlow's stateful dataflow graph.

We benefit from the event delivery guarantees offered by Flink or Beam. StateFlow relies *exactly-once* guarantee, ensuring an event is only processed *once* even in the case of failures. Inherent to the architecture of streaming systems, events with identical keys end up at the same logical operator and will be executed sequentially. Therefore, in the context of StateFlow, there is no concurrency for stateful entities with the same entity address. This is an advantage as concurrent updates to the same stateful entity might result in race conditions.

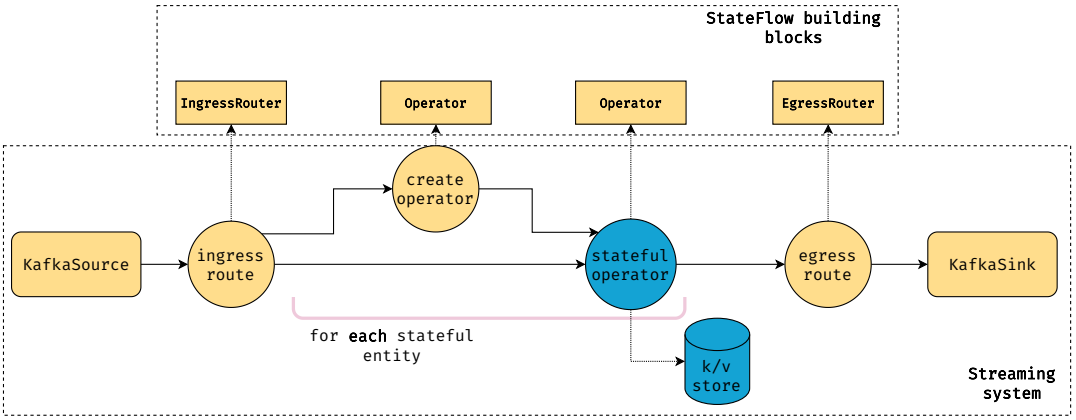


Figure 5.5: Proposed dataflow architecture embedding StateFlow execution blocks.

Figure 5.5 shows the general dataflow architecture used in Flink and Beam. Note that for **each** stateful entity we have a 'create operator' and a 'stateful operator'. In the running example, this means we have a create and stateful operator for `Item` as well as for `User`.

We distinguish between a 'create' and a 'stateful' operator to deal with the creation of stateful entities. Whenever an entity is instantiated, we cannot evaluate its `def __key__()` method before its constructor is called. Therefore, we handle a request to instantiate a new entity in the 'create operator'. This operator, given the arguments, calls the `def __init__( .. )` method. Afterward, we derive the key of the newly created entity and extract its state. Using the key, we traverse the streaming dataflow graph to end up at the correct operator. In the stateful operator, we store the state of the newly created entity. After entity initialization, the key is known, and we directly invoke all other class methods in the stateful operator.

Although we rely on execution via the streaming system, the actual logic is encapsulated in the building blocks. A major advantage of this approach is that application logic is not tightly coupled to the underlying system anymore. Figure 5.6 presents an alternative perspective on the proposed architecture. Here we show the actual processing graph for the Item operator with a parallelism of

2. For simplicity, we omitted the 'create operator' node and the StateFlow building blocks. All orange blocks in the architecture figures encapsulate *stateless* operations, whereas the blue blocks are *stateful*.

State is partitioned across all parallel instances (in Figure 5.6: 2 instances), and each stateful operator is responsible for a subset of this state. After routing, data is redistributed across the different parallel instances and forwarded to the correct stateful operator. In the case of StateFlow, each piece of state represents a stateful entity. Therefore, each instance of a stateful operator stores a subset of the stateful entities. For example, in the figure, one of the operator instances stores the Item entities with the keys: `jeans`, `t-shirt`, and `socks`. All events corresponding to those keys end up at this particular instance. Streaming systems ensure that the corresponding piece of state is available upon arrival of such an event. We pass this available state, and the arrived event to the StateFlow operator building block, which handles the execution. We update the resulting state in the stateful operator before forwarding the resulting event to the egress router. Finally, the figure also shows how we achieve this architecture through map, keyBy, and process operators in Apache Flink. We use a similar syntax for Apache Beam.

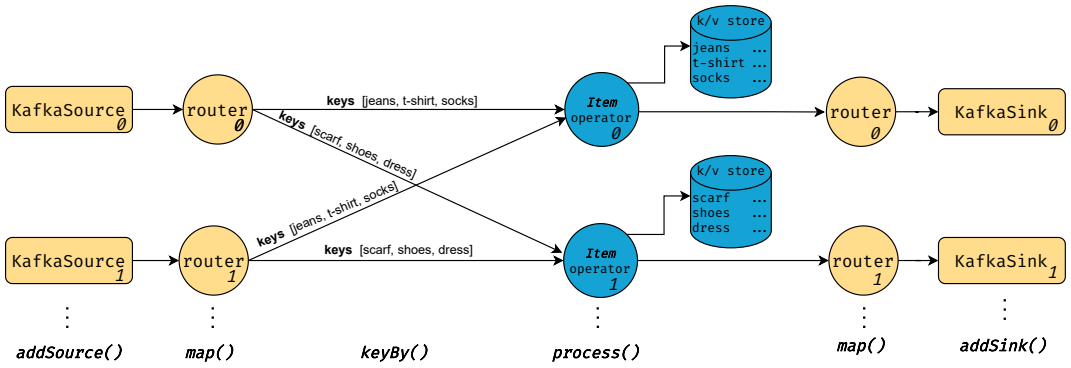


Figure 5.6: Processing graph for the **Item** operator with a parallelism of 2. For simplicity the 'create operator' node and the StateFlow execution blocks have been omitted.

As a bridge between the streaming system and a client side, we use *event streaming* in the form of Apache Kafka. Events are produced to a single *request topic* in an Apache Kafka cluster at the client side. An incoming event enters the streaming system via the Kafka source and leaves the streaming system again via a Kafka sink to the *reply topic*. An advantage of Kafka is that its parallelization strategy has seamless integration with streaming systems like Flink or Beam. Moreover, Kafka and the streaming system together offer the exactly-once guarantee. As streaming systems do not support cycles in their (directed and acyclic) streaming graph, we also use Kafka to cycle events back into the system. These cycles are necessary for split function invocations as such events need to move back and forth between different stateful entities located at different operators.

As mentioned before, StateFlow offers the building blocks for generating the streaming dataflow graphs using the intermediate representation. Therefore, the integration with frameworks such as Beam and Flink is lightweight. The integration code for Apache Flink (in Python) covers only **169** lines of code, whereas, for Apache Beam (in Python), this is **153** lines of code.

Flink JVM We propose a second integration using Flink in Java rather than Python. In this setup, we use a streaming system to enrich the events with the corresponding entity state and then do the actual (Python) execution on remote stateless functions. Figure 5.7 shows this architecture. The architecture is similar to the previously presented architecture, apart from the actual event execution. In this architecture, we execute the events in a remote stateless function. In the other architecture, we directly execute events in the operator that stores the corresponding state.

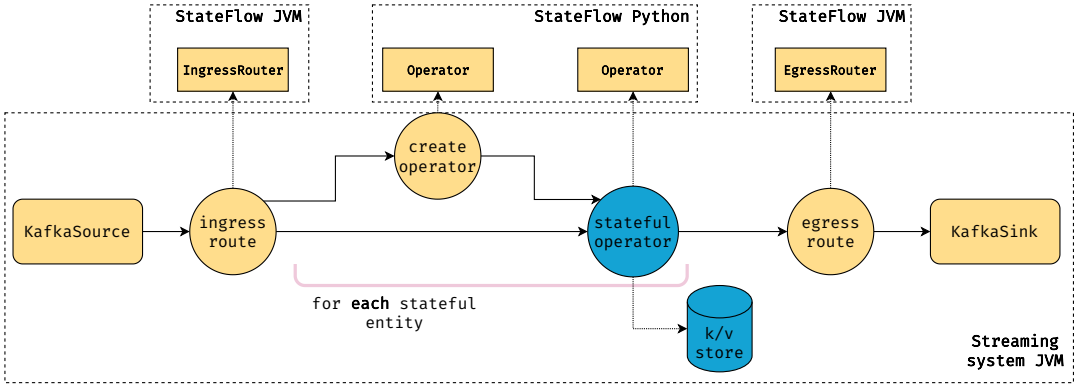


Figure 5.7: Dataflow architecture in which events are executed in remote stateless functions.

A significant advantage of this architecture is that we use the streaming system solely for routing and state enrichment. In other words, we use the streaming engine to route events, enrich them with the correct state, and then execute the event in a remote stateless (Python) function. Therefore the streaming architecture can run its native implementation, in this case in Java. This way, we benefit from the improved performance and maturity of the system. In addition, one can execute the Python code in a remote stateless executor of one's choice. We believe stateless functions in the cloud are a good choice, as they scale automatically, adopt a pay-as-you-go model, and are managed by a cloud provider. These functions are stateless and only initialized with the StateFlow operator building block.

For interoperability between the JVM and Python, we rely on Protobuf. We define a protocol buffer for StateFlow's events. This way, we can send and modify the events between Python and Java environments. Components like the execution graph, which we embed in the payload of an event, store specific Python constructs. We pickle (i.e., using pickle serialization) these into the event payload as *bytes* such that the Python components can unpickle and use it directly. Therefore, the payload of an event is not touched by any of the Java components.

In this architecture, we use Apache Flink as the streaming system and AWS Lambda for the (stateless) execution of the Python code. However, switching to another streaming system written in a JVM language like Apache Beam or Hazelcast Jet is also possible. Similarly, one can port the other remote stateless functions to execute the events, such as Azure or (Google) Cloud functions or even HTTP servers. To realize this integration, we ported the StateFlow router components to a Java implementation. Moreover, StateFlow generates a streaming dataflow graph given a list of stateful entity names. Given the StateFlow routing

components, supporting this architecture only spans **348** lines of Java code. The code for this architecture is part of a different repository [3].

5.2.2 Stateful Function-as-a-Service

Although we adopt a dataflow model for the stateful entities, we believe its execution is not limited to dataflow systems. We now propose a set of architectures, embedding the building blocks of StateFlow, using (S)FaaS systems. First, we propose an architecture that adopts stateless (cloud) functions — secondly, we show how to build on top of Flink Statefun. Finally, we present an architecture on top of CloudBurst Sreekanti et al. [2020], a popular research initiative on Stateful FaaS. We believe many (S)FaaS architectures can be supported with the existing StateFlow functionality. Therefore, the proposed architectures also serve as inspiration to support other (S)FaaS systems.

AWS Lambda AWS Lambda is a stateless FaaS solution and has no notion of state. Therefore, we rely on an external service for state storage. We opt for DynamoDB, a performant key-value store. DynamoDB is also an AWS service and follows the serverless model. Therefore it scales on-demand and follows the pay-as-you-go model. Although we use AWS Lambda also in the Flink JVM setup, its purpose is entirely different. AWS Lambda compromises only a tiny part of the complete architecture in the Flink JVM integration: it acts as a stateless Python executor. In this setup, AWS Lambda acts as a complete runtime handling all components such as routing and state management.

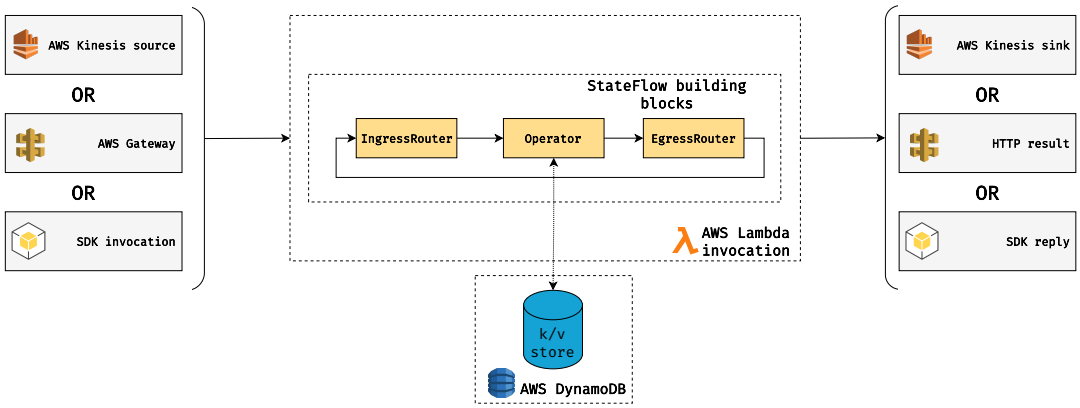


Figure 5.8: Execution architecture of stateful entities on top of AWS Lambda and AWS DynamoDB.

Figure 5.8 shows the architecture using AWS Lambda and DynamoDB. There are multiple ways to invoke an AWS Lambda instance with an event. StateFlow offers the three most common approaches: AWS Kinesis, AWS Gateway or invocation via the AWS SDK.

Each approach has its advantages and disadvantages. For example, AWS Kinesis allows event streaming and batch invocations rather than direct Lambda invocations. However, Kinesis has a higher latency compared to AWS Gateway

[3] https://github.com/wzorgdrager/stateflow-flink

or the SDK. AWS Gateway serves an HTTP interface in front of the AWS Lambda service, where the SDK invocation has the lowest latency but requires code.

Whenever an event arrives in the AWS Lambda invocation, its first passed to the ingress router. This router derives which stateful entity is involved. The state of this entity is requested from DynamoDB before StateFlow executes the event in its operator building block. We update the resulting state in DynamoDB and pass the resulting event to the egress router. As we have access to all the state in DynamoDB, we do not need to cycle events to the other operators as necessary for streaming systems. Instead, we loop these events back to the ingress router. We repeat this loop until the full execution graph is traversed, and we send the result back to the client. Therefore, we execute split functions in a single AWS Lambda invocation.

AWS Lambda and DynamoDB offer far fewer guarantees than streaming systems do. There is no notion of delivery semantics nor fault-tolerance. Using this architecture might result in events being lost or inconsistent state. Streaming systems guarantee that identical keys for the same operator (i.e., a single stateful entity) are executed in order rather than in parallel. Not doing so might result in race conditions. We do not have this guarantee in the AWS architecture. However, to simulate this in-order execution of identical keys, we use key locking. In other words, before the execution of the event, we 'lock' the corresponding key in DynamoDB. During this lock, no other AWS Lambda instance can use it. When the lock is obtained, the event is executed, and afterward the lock is freed again. As DynamoDB does not offer key locking as a feature, we rely on an external library. The key locking has implications for the performance on which we elaborate in Chapter 6.

Again, the proposed architecture primarily relies on the StateFlow building blocks. The integration with the AWS services is very lightweight and implemented in only **190** lines of code.

Flink Statefun Flink Statefun is a Stateful FaaS solution built on top of Apache Flink. Statefun revolves around the concept of a *stateful function*: an event handler having access to a partitioned piece of state. Statefun application consist of two parts 1) this stateless event handler are encapsulating business logic 2) a Flink cluster sending events and state to the stateless event handlers. More specifically, each incoming event has a particular key, the Flink cluster retrieves the state for that key, and finally, both the event and the state are sent to the function handler. In the stateful function handler, one can manipulate the (keyed) state, which is then persisted by the Flink cluster. Moreover, Statefun enjoys the many features of Flink such as exactly-once processing and state management.

These stateful functions are deployed as (remote) stateless HTTP servers to which a Flink cluster sends events and state. Below we show an example of an event function handler in Flink Statefun. This application computes a seen counter for users.

```
@functions.bind(
    typename='greeter',
    specs=[ValueSpec(name='seen_count', type=IntType)])
async def greet(ctx: Context, message: Message):
    name = message.as_string()

    storage = ctx.storage
    seen = storage.seen_count or 0
    storage.seen_count = seen + 1

    ctx.send_egress(kafka_egress_message(
        typename='greets',
        topic='greetings',
        key=name,
        value=f"Hello {name} for the {seen}th time!"))
```

In this function handler, an event arrives in the form of a message and state is stored in a context. Each handler is invoked based on a key, in this example the key is a *name*. Therefore, we also retrieve the seen count state of that belongs to that particular name. Notice, how this event handler is complete stateless. State manipulation on the context object is returned to the Flink cluster.

The execution in Statefun is very similar to the proposed Flink JVM architecture in the previous section. Nevertheless, it has three significant differences. First, the Flink JVM also supports stateless operations, for example, when creating a new stateful entity. Statefun requires every function call to be stateful. Secondly, Flink JVM's concept and integration code generalizes to other streaming systems, whereas Statefun tightly integrates with Apache Flink. Finally, Statefun implements its internal routing mechanism supporting cycles, whereas the Flink JVM setup cycles events via Kafka.

Figure 5.9 shows the integration of StateFlow with Flink Statefun. The figure also gives insight on the default workings of Statefun. For each operator (i.e., stateful entity) in StateFlow's IR, we generate a stateful function (handler) in Statefun. We rely on Python *closures* to generate these functions dynamically. A HTTP server runs these stateful function and multiple functions reside on the same server.

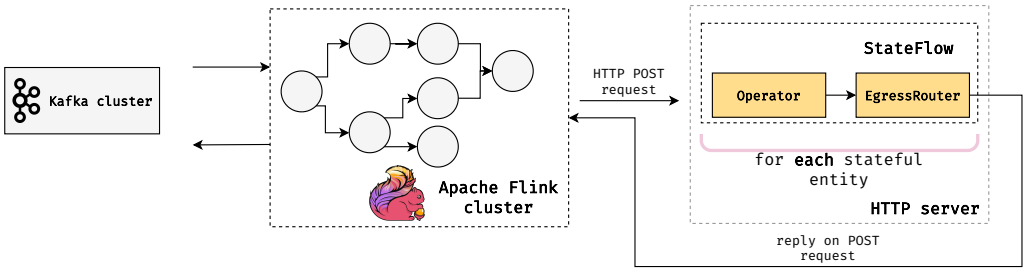


Figure 5.9: Integration of StateFlow with Flink Statefun.

These HTTP servers are stateless, and the Flink cluster triggers computation by calling an HTTP endpoint. In this call, Flink encodes the incoming event plus the state (i.e., the state of the corresponding stateful entity). After computation in the stateful function, the updated state and the outgoing event are sent back

to the client. Confusingly, Statefun calls these functions stateful, but in practice, these are stateless functions performing stateful computation.

In the streaming system integration, the client interface produces events to a single Kafka topic. Subsequently, the streaming framework routes to the correct operator using Stateflow's ingress router. For Flink Statefun, we cannot use a single topic as it requires a topic per stateful function. Therefore, we use a Kafka topic per stateful entity (i.e., a topic for the User entity). Moreover, we ensure each Kafka message, besides an event, stores the key of the stateful entity concerned. Therefore, we already route events at the client-side to determine the correct Kafka topic and message key. Unlike the streaming systems, Flink Statefun allows and takes care of messaging to other stateful functions.

As Statefun consists of two components, the HTTP server for the stateful functions and the Flink cluster, StateFlow offers integrations for both. We offer the HTTP server as a runtime implementation similar to Figure 5.4. Moreover, we generate the configuration required to deploy a Statefun application on a Flink cluster given a stateful dataflow graph. We present a full example below:

```
# Generate Flink configuration
statefun_module_generator.generate(flow)

# Setup and start runtime.
runtime: StateflowRuntime = StatefunRuntime(flow)
if __name__ == "__main__":
    web.run_app(runtime.get_app(), port=8000)
```

The proposed architecture scales in two ways. First, one can add more resources to the Flink cluster and increase its parallelism. Secondly, one can scale the number of remote function deployments. These deployments are stateless and can be scaled horizontally. It requires a load balancer in front of these deployments to distribute the workload over the HTTP deployments. Most cloud providers offer these load balancers as a service.

The integration of StateFlow with Flink Statefun spans **121** lines of code.

CloudBurst CloudBurst is a Stateful FaaS solution with a dedicated runtime built on top of the Anna key-value store Wu et al. [2018]. Like Flink Statefun, developers implement stateful functions, and the runtime takes care of scaling, fault-tolerance, and state management. Moreover, CloudBurst allows function-to-function communication and a client-side interface to invoke the stateful functions. We show the full CloudBurst architecture in Figure 5.10.

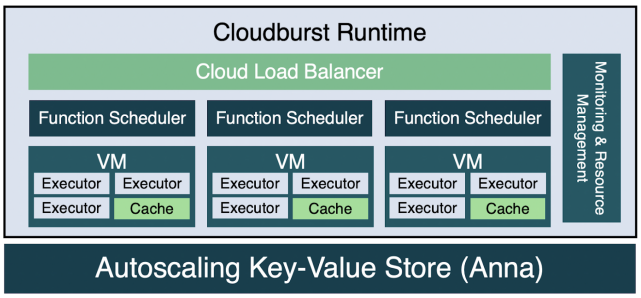


Figure 5.10: The complete CloudBurst architecture. Image retrieved from Sreekanti et al. [2020]

.

In Figure 5.11, we show the integration of StateFlow with CloudBurst. We build a CloudBurst function for each building block in StateFlow: an ingress router, egress router and operators for each stateful entity. In these functions, we have direct access to the Anna key-value storage. Therefore, in the operator function, we directly query this store to get and update the state of entities.

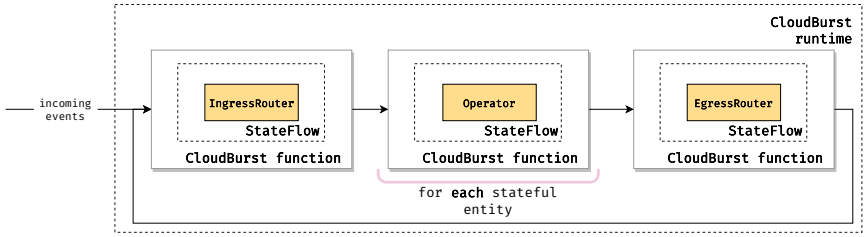


Figure 5.11: Integration of StateFlow with CloudBurst.

The integration of StateFlow with CloudBurst spans **87** lines of code. Unfortunately, we could not get the CloudBurst runtime working in a local setup nor a cluster [4]. The project seems to be abandoned, and some artifacts, like required Docker images, are deleted. As a result, we decided not to add a client-side implementation for CloudBurst in the StateFlow prototype.

5.3 Client

Although the runtime systems perform the heavy-lifting in terms of computation, clients trigger these computations in the form of events. In other words, a runtime executes a class method, but the client triggers this execution. The primary goal of a client is to be a lightweight interface to the runtime system. Similar to the runtime systems, clients rely on the intermediate representation compiled by StateFlow. Amongst others, the client uses this IR to generate the correct events.

StateFlow offers a generic client-side programming interface, which allows developers to interact with the deployed stateful entities. We elaborate on this interface in Subsection 5.3.1. This programming interface integrates with multiple clients, such as Apache Kafka, and enables communication with the runtime system. Essentially, each of these clients provides an event streaming integration, and we discuss these in Subsection 5.3.2. Finally, StateFlow offers an HTTP

[4] Other TU Delft students raised several issues in the CloudBurst GitHub repository without any luck: https://github.com/hydro-project/cloudburst/issues

integration on top of each client such that each stateful entity can be manipulated via HTTP endpoints. We present this integration in Subsection 5.3.3.

5.3.1 Interface

In order to improve the developer's experience, StateFlow client-side interface is close to the programming experience in a local Python environment. In general, the client-side interface allows developers to instantiate objects and call their methods. In the background, StateFlow generates events for each action and sends them to the runtime. The resulting events are asynchronously retrieved in a background thread before, using a future construct, StateFlow reports the result.

To achieve this, the `@stateflow` decorator intercepts a class definition and replaces it with a **metaclass**. Using metaclasses is a form of metaprogramming in Python and gives us control over class creation. This metaclass stores all analyzed metadata of a class definition, as derived from the compiler pipeline (Chapter 4). Upon construction of this substituted metaclass, for example `Item(10, "jeans")`, StateFlow sends a `InitEntity` request to the runtime. If successfully created, the metaclass constructs a new wrapper object for that specific entity. A wrapper object stores a reference to a stateful entity in the form of an entity address (see Subsection 5.1.1). Using this wrapper is a form of **duck typing**: from a developer perspective it seems like one interacts with the actual object. On the other hand, StateFlow wraps results in futures as communication with the runtime is asynchronous. For existing entities, StateFlow's offers utility methods to generate a wrapper object: `item: Item = Item.from_key("jeans")`. A full example of client-side interaction is shown below:

```python
# entities.py stores Item and User class definition
from entities import User, Item, stateflow
from stateflow.client.client import StateflowClient, StateflowFuture

# Initialize client
client: StateflowClient = ...

user: User = User("test-user").get()

# create 10 items, concurrently
items_fut: List[StateflowFuture[Item]] = \
    [Item(25, f"test-item-{i}" for i in range(0, 10)]
items: List[Item] = [fut.get() for fut in items_fut]


# add items to basket of user
user.add_to_basket([items]).get()
```

Whenever a method is invoked, StateFlow generates the corresponding event. Each event carries a unique id, which we use to trace back reply events to its requests. Moreover, StateFlow extracts the entity address from the wrapper object. Finally, we embed the method call arguments into the events payload. Whenever

a split function is invoked, we copy its execution graph into the event. Moreover, we set the call arguments in the output variables of the start node (see Subsection 4.3.6). Besides method invocation, StateFlow also supports setting and getting entity attributes:

```
user: User = User("test-user").get()

# Get the balance
the_balance: StateflowFuture[int] = user.balance
the_balance.get()

# Set the balance
user.balance = 0
```

A downside of setting attributes is that updates are asynchronous, whereas the syntax gives the impression that it is not. One cannot use StateFlow's future system with attribute assignments.

Asyncio As an alternative to StateFlow's futures, one can use Python asyncio. Asyncio is the default concurrency library for Python and allows writing asynchronous code through the `await` and `async` keywords. The code snippet below shows asyncio code for StateFlow:

```
user: User = await User("test-user")

await user.set_balance(10)
balance = await user.get_balance()
```

Asyncio does not allow setting or getting attributes asynchronously (e.g. `user.balance`); instead, one needs to define getter and setter methods explicitly. The use of asyncio requires an asynchronous implementation of a client such as Apache Kafka. In Section 5.6, we highlight which clients have this integration.

5.3.2 Event streaming

To bridge between client and runtimes, we rely on event streaming platforms. These platforms act as middleware between the client and the runtime and have several advantages. Most significantly, these platforms provide a durable event stream and are highly scalable. This middleware scales with the application's load and ensures no event is lost in case of failures. Additionally, these platforms handle backpressure whenever the runtime cannot keep up with the requests from a client. Most event streaming platforms support *at-least-once* delivery for events, whereas others support even stronger consistency, such as *exactly-once* delivery. These consistency models ensure that client-runtime communication is reliable.

StateFlow supports two event-streaming clients: Apache Kafka and AWS Kinesis. Apache Kafka integrates with runtimes such as Apache Flink, Apache Beam,

and Flink Statefun, whereas AWS Kinesis integrates with the AWS Lambda runtime. AWS Kinesis only supports *at-least-once* delivery, whereas Apache Kafka supports *at-least-once*, *at-most-once*, and *exactly-once*. Note that these platforms run as a separate deployment. To use Apache Kafka requires a separate Kafka cluster. AWS Kinesis is offered as a service by Amazon and requires deployment on their cloud platform. The clients integrate into StateFlow by sending and receiving events (i.e., publish and subscribe) to and from these platforms, whereas runtime systems have a similar integration.

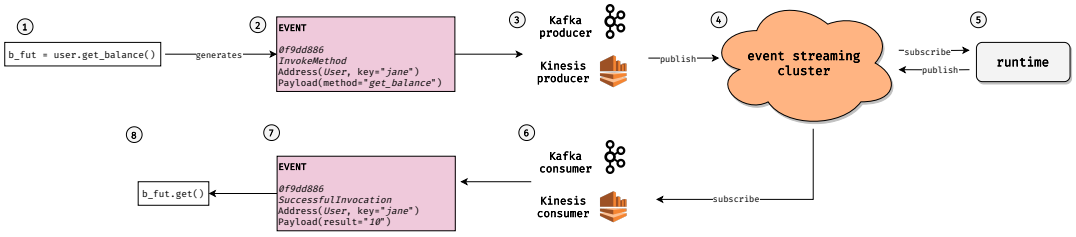


Figure 5.12: StateFlow's client integration with event streaming platforms.

In Figure 5.12, we show how StateFlow integrates with these platforms. Upon method invocation (1), StateFlow generates the corresponding event (2) which is abstracted away from the developer. Via a Kafka or Kinesis producer (3), the event is published to its cluster (4). The runtime system reads the event from the cluster, processes it, and sends the resulting event back to the event streaming cluster (5). In a separate thread, the client receives all incoming events via its consumer (6). The resulting event, with the identical event id, is parsed (7), and the corresponding future is enriched with the results (8).

Not all runtimes support each event streaming platform. In addition, some setups completely omit such a platform and directly send events from a client to a runtime. In Section 5.6, we highlight all client integrations and their compatibility with runtimes.

5.3.3 REST API integration

REST is one of the most popular web API architectures. For thit reason, StateFlow supports a REST integration on top of several of the presented clients. Such a REST API allows interacting with the deployed stateful entities via HTTP endpoints without writing client-side code. StateFlow leverages FastAPI, a modern web framework, to create such a REST API [5].

With the help of the intermediate representation, StateFlow dynamically creates endpoints for each class method, entity construction, and entity lookup. For example, to find an Item stateful entity with the key `jeans`, one queries `/stateflow/global/Item/find?key=jeans`. Upon receiving such a query, StateFlow constructs the corresponding event, sends it via the client to the runtime, waits for the runtime's return event, and, finally, provides an HTTP response. In FastAPI, one defines HTTP endpoints via function definitions, and StateFlow generates these functions automatically. In Figure 5.13, we show the automatically generated REST API for the entities in the running example.

[5] FastAPI was chosen out of convenience and could be substituted by any other web framework.

Figure 5.13: Automatic generation of HTTP endpoints for all example stateful entities in FastAPI.

For the endpoint generation, StateFlow transforms the parameters of a Python method to query parameters of an HTTP endpoint. Primitive types can be expressed as a string, and therefore its conversion is straightforward. However, we cannot do the same for complex types like objects or lists. We convert object parameters to a string and a list of objects to a list of strings to tackle this problem. This requires that the types of these objects are other stateful entities annotated in StateFlow. If that is not the case, we abort the endpoint generation for that method and give a warning message. For example, consider the following function signature in the `Item` stateful entity: `def foo(self, x: int, y: User, l: List[User])`. An example query on this endpoint: `stateflow/global/Item/foo?x=1&y=jane&l=joe&l=anna`

In addition to the generated endpoints, developers can add more endpoints encapsulating custom behavior. For that, one uses the FastAPI interface in combination with StateFlow's asyncio integration. For example, to create a GET endpoint that creates multiple entities of the type `Item`:

```python
@app.get("/create_multiple_items")
async def create_items(item_names: List[str]):
    for item in item_names:
        await Item(0, item) # Create item, set price to 0.

    return "Created all items!"
```

5.4 Local execution

To improve transparency and usability, StateFlow offers a local execution environment to unit test all defined classes. This execution environment simulates both the client and runtime environment. In the code snippet below, we show how one uses this environment.

```python
from stateflow import stateflow_test

def test_add_to_basket():
    user = User("test-user")
    item = Item(10, "test-item")

    assert user.balance == 1000
    assert user.add_to_basket([item]) is True
    assert user.basket[0].itemid == "test-item"
```

From a developer's perspective, this is a standard unit test. However, underneath, the environment creates events, passes them through all building blocks, and returns a resulting event. Instead of relying on a runtime, the environment stores all the entity state in a local hashmap. We use metaprogramming to intercept object construction and inject the local execution environment.

An advantage of this local execution environment is that execution follows all steps used in client and runtime environments. For example, when a function is split, it moves back and forth between different operators. This local environment ensures that, when local execution succeeds, results will be similar in a distributed setup using a runtime.

5.5 Deployment

So far, StateFlow has introduced several integrations with runtimes, clients, and even an HTTP server. Nevertheless, we did not yet cover a costly operational aspect of (cloud) applications: deployment. Each of the involved components requires its own configuration and deployment procedure. For example, to deploy an AWS Lambda runtime, one needs to package the application, including its dependencies, and upload it to the AWS platform. In addition, one needs to set up a DynamoDB cluster and configure permissions such that each service has access. Contrarily, Apache Flink requires deploying a standalone Flink cluster, packaging the application, and submitting the application via a separate CLI tool.

To support the deployment process, StateFlow offers several default deployment configurations and tools. First of all, for all standalone deployments, such as the Apache Flink cluster or the FastAPI web server, we propose *Kubernetes* manifests [6]. Kubernetes is the most popular container orchestration platform allowing deploying and scaling of container applications. These manifests allow a one-click deployment of any application and provide a starting point for more tailored deployments. Secondly, for AWS Lambda, we provide a configuration for *serverless* [7]. Serverless is a toolbox to package, configure and deploy AWS Lambda applications automatically. Finally, for Flink Statefun, one needs to create an application configuration that defines all involved functions, inputs, and outputs. Apache Flink uses this configuration to create a corresponding streaming dataflow graph. Since creating this configuration is a painstaking activity, StateFlow offers a utility to automatically generate this configuration file, given the intermediate representation of the implemented application.

[6] https://kubernetes.io/
[7] https://www.serverless.com/

Besides default configurations and deployment files, StateFlow has a deployment guide for each client and runtime. By these offerings, StateFlow's attempts to provide an 'end-to-end experience'. In other words, we support developers in implementing, configuring, and deploying their applications.

5.6 Summary

In this chapter, we presented several clients and runtimes for the execution of applications in StateFlow. We summarize all implemented clients in Table 5.2 and show if they integrate with acyncio and our HTTP integration. Table 5.3 shows all implemented runtimes and their compatibility with the clients. In addition, we show which runtimes do not support execution in clusters or locally. For example, CloudBurst suffers bugs which prevents execution at all.

Client name	Service	asyncio integration	HTTP integration
`KafkaClient`	Apache Kafka	✔	×
`AWSKinesisClient`	AWS Kinesis	×	×
`AWSGatewayClient`	API Gateway	×	×
`KafkaFastAPIClient`	Apache Kafka	✔	✔
`AWSGatewayFastAPIClient`	API Gateway	✔	✔
`AWSLambdaFastAPIClient`	AWS SDK	✔	✔

Table 5.2: Summary of StateFlow's supported clients and its integrations.

Runtime name	Service	Local execution	Cluster execution	Compatible clients			
				Apache Kafka	AWS Kinesis	API Gateway	AWS SDK
`FlinkRuntime`	Apache Flink (PyFlink)	✔	✔	✔	×	×	×
`StatefunRuntime`	Flink StateFun	✔	✔	✔	×	×	×
`AWSLambdaRuntime`	AWS Lambda	✔	✔	×	×	×	✔
`AWSGatewayLambdaRuntime`	AWS Lambda	✔	✔	×	×	✔	×
`AWSKinesisLambdaRuntime`	AWS Lambda	✔	✔	×	✔	×	×
`BeamRuntime`	Apache Beam	✔	×	✔	×	×	×
`CloudBurstRuntime`	CloudBurst	×	×	×	×	×	×
✱ `RemoteLambda`	Apache Flink (JVM)	✔	✔	✔	×	×	×

Table 5.3: Summary of StateFlow's supported runtimes.

✱ The Flink JVM runtime requires the deployment of an AWS Lambda Python application *and* a (JVM) Flink application.

Chapter 6

Evaluation

In this chapter, we evaluate the work of this thesis from three angles. Firstly, we evaluate the *expressiveness* of StateFlow's programming model compared to the native programming model of each runtime (Section 6.2). Secondly, we show the *overhead* of all the components in StateFlow. That is, showing how much overhead our system incurs on top of executing the Python code written by the developer (Section 6.3). Moreover, we show how this overhead relates to the overhead of the underlying runtimes. Thirdly, we execute a *performance* benchmark with the different runtime backends supported by StateFlow, including AWS Lambda, Apache Flink, Flink Statefun, and Flink JVM (Section 6.4). In this experiment, we show how each of the runtimes performs under an increasing workload. In the expressiveness and performance experiments, we rely on the DeathStar benchmark Gan et al. [2019]. We explain this benchmark in Section 6.1. Finally, we use the following runtime versions for all experiments: Apache Flink and PyFlink 1.13.0 and Flink Statefun 3.0.

All code for the experiments in this chapter can be found in an open-source repository [1]. To improve replicability of all experiments, we explain the experimental setup in detail, and Jupyter Notebooks and Python scripts[2] are provided to show how results are post-processed. Moreover, each section in the Jupyter Notebook/code links to the implementation used for the corresponding experiment.

6.1 DeathStar benchmark

For expressiveness and performance evaluation, we opt for the DeathStar benchmark Gan et al. [2019]. Popular benchmark alternatives include YSCB Cooper et al. [2010] and TPC-C Raab [1993]. However, both these alternatives primarily focus on transactional workloads, and the uses cases are synthetic. Contrary, DeathStar introduces a set of real-world applications and its microservice design. These applications include a social media network, a banking system, and a hotel reservation service. For these experiments, we only focus on the hotel reservation service and port its implementation in StateFlow. Whereas DeathStar provides Go code for most of their services, specifications are not well-defined. Therefore, we specify the requirements of each service using the information provided in both the paper and the open-source code. One downside of DeathStar is that it is designed for microservices. In StateFlow, we partition state on an entity key and

[1] https://github.com/delftdata/stateflow-evaluation
[2] An example notebook

distribute these entities among different instances. However, a microservice architecture assumes access to a (global) database storing all the state. DeathStar also incorporates this assumption into its design. Therefore, we make some slight adjustments in its design to make it fit the stateful entities philosophy. To the best of our knowledge, there does not exist a benchmark for partitioned stateful entities.

Hotel reservation service As the name suggest, this application mimics that of a hotel reservation service. It includes *seven* stateful entities and *four* endpoints. We implement the entities as Python classes and the endpoints in the HTTP integration of StateFlow. We give a full overview in Figure 6.1. Below we elaborate on each stateful entity and its functionality:

- **Geo**: determines the five closest hotels given a latitude and longitude coordinate.
- **Rate**: returns the rate plans for a list of hotel ids.
- **Search**: given a latitude and longitude coordinate, determines the five closest hotels and return their rate plan. This entity interacts with the Rate and Geo entities.
- **Recommend**: recommends a list of hotel ids. A recommendation has either the shortest distance, the lowest room price, or the highest hotel rate.
- **Profile**: returns the profiles for a list of hotel ids.
- **Reserve**: either reserves a hotel room or checks the availability for a specified date. This entity represents the reservation service of a specific hotel.
- **User**: logins a user, given the correct password.

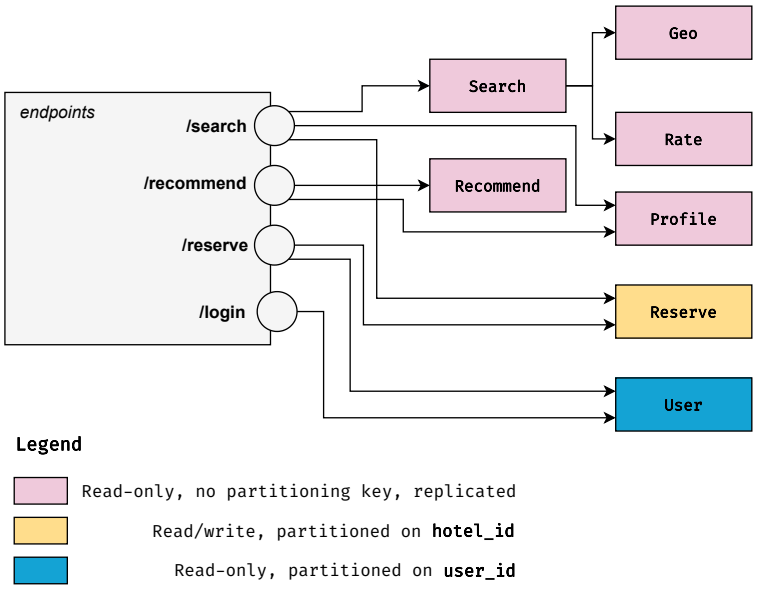


Figure 6.1: Schematic overview of DeathStar's hotel service implementation in StateFlow.

As mentioned before, DeathStar is designed as a microservice benchmark, and assumes some of its services have access to the global state. In the figure, all these services are in pink blocks. In StateFlow, we do not have this notion of a global state. Instead, we always partition entities on their key, and requests for equal keys are processed atomically. Consequently, if we have only a single instance of such an entity, the system does not scale. To overcome this issue, we replicate the read-only entities. For example, we create n copies of the Geo entity, all with a unique key. This way, we can have n concurrent requests querying the Geo service. In other words, we mimic a read-only DeathStar microservice with n replicated stateful entities. Such a strategy cannot be used with services requiring write requests, as entities would have to synchronize their data. At the same time, there is a straightforward partitioning key for both the User and Reserve entity. For the User entity, this is the unique user id, whereas, for the Reserve entity, this is the hotel id. The number of instances for these entities equal respectively the amount of users or hotels.

Below we summarize the different endpoints as implemented in StateFlow HTTP integration. For the read-only replicas, these endpoints select a replica at random. We assume that over time and as the throughput scales, replicas are selected uniformly, distributing the load.

- **/login**: logins a user (User entity).
- **/recommends**: first requests a set of recommendations (Recommend entity) and then retrieves the profiles of the recommend hotels (Profile entity).
- **/search**: requests a search (Search entity) which returns a set of hotel ids. For each hotel, availability is checked (Reserve entity). Finally, for all available hotels profiles are requested (Profile entity). In total, this endpoint involves **nine** stateful entity calls.
- **/reserve**: logins a user (User entity) and a reserves a hotel room (Reserve entity).

In the expressiveness evaluation (Section 6.2), we show how the original DeathStar Go implementation relates to that of StateFlow. Moreover, we compare native runtime implementations of DeathStar to implementations using StateFlow. Finally, we use DeathStar in the performance experiments (Section 6.4) to evaluate the runtimes under an increasing load.

6.2 Expressiveness

In this section, we discuss the expressiveness of the programming interface of StateFlow. We show that developers only have to focus on application code rather than non-application code such as infrastructure, event routing, serialization, and storage. We discuss this using the implementation of the DeathStar benchmark (Section 6.1). First, we show how StateFlow's implementation differs from the original implementation in the Go programming language. Afterward, we compare the StateFlow implementation to a native implementation in the target runtimes.

6.2.1 StateFlow versus DeathStar

In Table 6.1, we compare the StateFlow (Python) implementation to the original implementation of DeathStar in Go. The table shows the lines of code (LOC) for each stateful entity and the percentage non-application code (NAC). For StateFlow we consider the amount of non-application code to be fixed: developers annotate their classes with `@stateflow` and implement a `def __key__()` method.

	StateFlow	**DeathStar Go**	
	LOC	LOC	% NAC
Search	13	33	0%
Geo	25	53	15%
Profile	26	64	20%
Rate	24	71	30%
Recommend	48	88	27%
Reserve	36	169	46%
User	9	14	0%

Table 6.1: Lines of code for the DeathStar implementation in StateFlow and the original in Go.

From the figure we conclude that DeathStar in StateFlow requires less than 50% lines of code for all stateful entities compared to the original implementation. Note that the original implementation is in Go, and therefore syntax is slightly different. In addition, in the DeathStar Go implementation, infrastructure code mixes with application code. More specifically, connections with databases, caches, and serialization methods are part of the application code. This leak from the infrastructure to the application layer has two significant disadvantages. First, code becomes more lengthy and complex and is therefore prone to bugs. Secondly, changing the underlying infrastructure requires refactoring the entire application. For example, changing the type of database requires a refactor for the whole class.

6.2.2 Native runtime implementation

We now discuss the StateFlow stateful entity implementation compared to native implementation in the target runtime. We compare implementations in AWS Lambda, Flink Statefun, and PyFlink. The Flink JVM and Apache Beam implementations would be very similar to that of PyFlink, and therefore, we omit it. All these runtimes offer a Python API. We focus on the stateful entities and do not compare code for the frontend or initialization functionality.

In Figure 6.2 we compare the StateFlow implementation for a User to a native implementation in AWS Lambda. We highlight the application code with pink. Similar to the Go implementation, AWS Lambda mixes the application with infrastructure code. For example, we have to store the user state in DynamoDB manually. In StateFlow, this is abstracted away from the developer. Moreover, in AWS Lambda, developers implement an event handler and one needs to write the logic for event parsing. In StateFlow, this is also taken care of, and developers implement functionality through class methods. Note that the User entity is

```python
@stateflow
class User:
    def __init__(self, username: str,
                 password: str):
        self.username: str = username
        self.password: str = password

    def login(self, password: str) -> bool:
        return self.password == password

    def __key__(self):
        return self.username
```

```python
dynamodb = boto3.client("dynamodb")
table = dynamodb.Table("users")
def user_handler(event, context):
    username = event["username"]
    pw = event["password"]
    if event["type"] == "CREATE_USER":
        user_item = {username: username, pw: pw}
        table.put_item(Item=json.dump(user_item))

        return {"message": "created a user!"}
    elif event["type"] == "LOGIN_USER":
        try:
            response = table.get_item(Key=
                {"username": username})
        except ClientError as e:
            return {"message": "user not found"}
        else:
            user = json.loads(response["Item"])
            return {"message": user["pw"] == pw}
```

Figure 6.2: User entity in StateFlow versus its native implementation in AWS Lambda. We highlight application code with pink.

still one of the simplest entities, whereas more complex ones like Reserve require even more routing, serialization, and database code. In this figure, we highlight AWS Lambda, but the code for Flink Statefun and PyFlink looks very similar. For PyFlink, developers must also implement a streaming dataflow graph to route events to the correct stateful operator.

	StateFlow	**AWS Lambda**		**Flink StateFun**		**PyFlink**	
	LOC	LOC	% NAC	LOC	% NAC	LOC	% NAC
Search	13	13	62%	20	55%	22	68%
Geo	25	31	32%	34	35%	39	48%
Profile	26	33	36%	35	37%	37	46%
Rate	24	30	43%	36	50%	43	52%
Recommend	48	54	26%	53	17%	54	18%
Reserve	36	60	48%	66	25%	61	35%
User	9	17	64%	22	50%	25	72%

Table 6.2: Comparison of lines of code for the DeathStar implementation in StateFlow and native runtime implementations.

In Table 6.2 we compare all entities in StateFlow to its native runtime implementation [3]. In these implementations, we keep application code as similar as possible. Similar to the comparison with Deathstar Go, we show how many lines of code (LOC) each entity requires and the percentage of non-application code (NAC). The table shows that, for all entities, the code footprint in StateFlow is smaller compared to native implementations. This difference is no surprise as StateFlow requires a minimal amount of non-application code.

[3] Native runtime implementations are available in the evaluation repository: https://github.com/delftdata/stateflow-evaluation/tree/main/alternative

Client side interaction All runtimes discussed in this thesis are event-driven. Therefore, developers implement event generation and handler code to interact with the runtimes on the client-side. In the User entity example (Figure 6.2), to interact with this AWS Lambda handler, one has to create an event like this: `event = {type: "LOGIN_USER", ..}` and send it to the runtime via the AWS SDK. In addition, one needs to implement code to handle the resulting events. In StateFlow, client-side interation with the runtimes is not event-driven. Instead, developers write (asynchronous) object-oriented code. For example:

```
# Create a user
user = await User("tudelft", "pass")

# Login a user
can_login = await user.login("tudelft", "wrong_pass")
```

We argue that, from an application perspective, such implementations are more intuitive than event-driven code.

Runtime portability One of the strengths of StateFlow is its portability. One can switch between runtimes rather easily and it does not require any changes to the application code. Switching between runtimes always requires less than 10 lines of code. In contrast, native implementations are tightly bound to their underlying runtime and switching runtimes requires heavy refactoring.

Moreover, implementations to support new runtimes are lightweight. With the help of StateFlow building blocks, we manage to reduce the amount of integration code. The interface implementation for using a new runtime system with StateFlow ranges between 87 and 190 lines of code for all supported runtime systems. Therefore, we argue that adding new runtimes is straightforward.

6.3 System overhead

In these experiments, we identify the overhead caused by using StateFlow as a programming model and as an integration with a runtime system. We show how overhead changes for different conditions in stateful entities. Moreover, we compare the overhead of StateFlow with that of the different runtimes. For this evaluation, we do not rely on the Deathstar benchmark and implement a synthetic workload. In StateFlow, we identify **five** distinct components which incur overhead. These have a clear connection with the StateFlow building blocks (Subsection 5.1.2).

1. **State serialization**: StateFlow serializes the state of an entity before storing it in a persistent database or persistent state storage of the runtime. Before and after storage, StateFlow serializes the state.

2. **Event serialization**: StateFlow serializes events when communicated over a network. As a result, events need to be deserialized when entering the runtime and serialized when it leaves the runtime.

3. **Routing**: In several stages of the dataflow, StateFlow routes events towards the next operator or stage. StateFlow offers a router for incoming events, the ingress router, and a router for outgoing events, the egress router.

4. **Stateful entity construction**: The lifetime of a stateful entity is limited to its invocation. Whenever a stateful entity is invoked, we first reconstruct the entity using its (persistent) state. After construction, the correct method is invoked, and finally, StateFlow retrieves the updated state and destructs the instance. We consider all these actions to be overhead, apart from the actual invocation.

5. **Execution graph traversal**: Methods of stateful entities might be interactive: they call methods of other stateful entities. StateFlow splits these methods in its compiler pipeline, and we execute such invocations by traversing an execution graph. Whenever a function is invoked, StateFlow traverses the execution graph and updates the nodes with output variables. Moreover, the graph is traversed backwards to find previously defined variables. Both types of traversal incur overhead.

We perform all the overhead experiments at the event level. In other words, we compute the overhead for an event entering the runtime system until it leaves the system again. We follow the event handling logic as explained in the introduction of Section 5.1 Note, the actual execution of a user-defined functionality does not count towards this overhead. In general, we compute the overhead as follow:

$$\texttt{overheadStateFlow} = \texttt{overheadBuildingBlocks}$$
$$\texttt{overheadRuntime} = \texttt{event}_{out} - \texttt{event}_{in} - \texttt{overheadStateFlow}$$

We consider the overhead of StateFlow, as the sum of durations for executing all building blocks involved in the event handling. The runtime overhead is the time difference between an event entering ($\texttt{event}_{in}$) and leaving ($\texttt{event}_{out}$) the system, minus the StateFlow overhead. For some runtime systems, we can give a more fine-grained analysis of where overhead is spent. For example, in AWS Lambda we show how much time is spent on database interaction.

We split the overhead evaluation into two parts: **without** runtimes and **with** runtimes. In the evaluation without runtimes, we evaluate the absolute duration of the different StateFlow components. For the evaluation with runtimes, we compare the overhead of StateFlow with that of the runtimes. All workloads for these experiments are synthetic. We design three types of experiments in which we vary with properties of stateful entities: **state size**, **execution graph length** and **interactivity**.

State size The state size is the amount of data that is stored inside the instance of a stateful entity. For our experiments we tests with sizes 50KB, 500KB, 5MB and 50MB. An example of such an stateful entity can be found in Figure 6.3. We simulate the state size by having a byte array of a fixed length.

Execution graph length Whenever StateFlow splits a function, we encode the execution behavior in an execution graph. This execution graph grows as the function has more remote function calls *or* control flow elements. This graph

```python
@stateflow.stateflow
class Entity50KB:
    def __init__(self):
        self.data = bytearray([1] * 50000)

    def execute(self):
        pass

    def __key__(self):
        return "entity50kb"
```

Figure 6.3: Stateful function with a state size of 50KB.

```python
@stateflow.stateflow
class EntityExecutionGraph10:
    def __init__(self):
        self.data = bytearray([1] * 50000)

    def execute(self,
        other: "EntityExecutionGraph10"):
        # Adding 'other' parameter which
        # triggers the function to be split.
        x = 1

        if True:
            pass
        ...
        if True:
            return x

    def __key__(self):
        return "entityexecutiongraph10"
```

Figure 6.4: Stateful function with an execution graph of a specific length.

is traversed backward to find previously defined variables and therefore incurs overhead. In practice, a function is only split when it has one or more function calls. However, in this experiment, we create a non-interactive function that has an execution graph. We can still force the compiler pipeline to split a function by adding a stateful entity as a parameter. To increase the length of the execution graph, we add more control flow elements as these elements trigger a split. By having a non-interactive (i.e., no remote function calls) execution graph, execution is limited to a single stateful entity, and we isolate the impact of an increasing execution graph. We show an example in Figure 6.4. We experiment with various execution graph lengths: 10, 50, 100, and 200. These execution graphs are acyclic. Finally, we fix the state size to 50KB.

Interactivity In this experiment, we vary with the interactivity of a stateful function. That is, the amount of remote function calls within a function. More specifically, we create a function that calls another stateful entity in a for-loop. In other words, the execution graph of this function is cyclic, and we vary with the number of cycles. Note that for these graphs, we move back and forth between different stateful entities different from the 'execution graph length' experiments which are limited to a single entity. We do not have a nested execution graph, as the remote functions are not split. An example of such an entity is shown in Figure 6.5. We vary with 5, 10, 15, and 20 interactions with other stateful entities. Again, we fixed the state size to 50KB for each entity involved.

```python
@stateflow.stateflow
class EntityInteractive:
    def __init__(self):
        self.data = bytearray([1] * 50000)

    def execute(self,
                others: List[EntityRemote]):
        for other in others:
            other.execute(other)

        return 1

    def __key__(self):
        return "interactive-entity"
```

Figure 6.5: Stateful entity which interacts with other stateful entities in a `for` loop.

To measure overhead, we synchronously invoke the `def execute(self)` method on the different stateful entities. This resembles sending *one* event to the runtime. Then for each event, we measure the overhead of all different components. We summarize all experiments types in Table 6.3 and repeat these experiments *with* and *without* runtime respectively in Subsection 6.3.1 and 6.3.2.

Experiment	Short description	Parameters	Code
State size	We compute overhead for various state sizes and a non-interactive stateful entity.	State size is 50KB, 500KB, 5MB or 50MB. There is no execution graph involved.	Figure 6.3
Execution graph length	We compute overhead for various execution graph lengths with a non-interactive stateful entity.	State size is fixed to 50KB. Execution graph length is 10, 100, 500 or 1000.	Figure 6.4
Interactivity	We compute overhead for various interactions with other stateful entities.	State size is fixed to 50KB. We experiment with 5, 10, 15 and 20 interactions.	Figure 6.5

Table 6.3: Overview of all experiment type for the overhead evaluation.

6.3.1 Overhead *without* runtimes

The goal of these experiments is to show the performance of StateFlow regardless of the underlying runtime. Moreover, we show how specific components of our system become more expensive as conditions of the stateful entities change.

Experimental setup

We execute the experiments as shown in Table 6.3 and measure the absolute duration, in milliseconds, of each component in StateFlow. We perform all computation in the local runtime (Section 5.4), which means that we store state in-memory. Both state and event serialization use the pickle serializer. We perform all these experiments on a local machine with an Intel Core i5-6600k CPU @ 3.5GHz and 32GB of RAM. To deal with performance variability, we repeat experiments 10000 times.

Results

The results for the non-interactive stateful entity with varying **state size** experiment can be found in Figure 6.6. For the duration (y-axis), we use a logarithmic scale as the durations span over a wide range. This figure shows the absolute

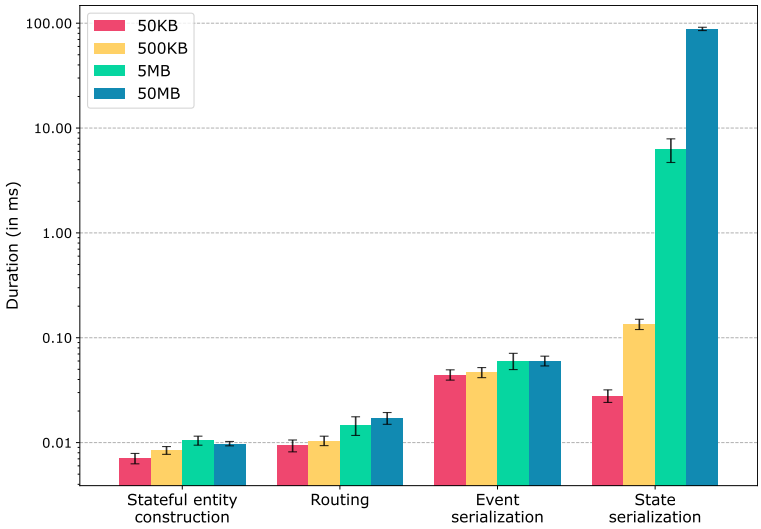


Figure 6.6: Duration of components in StateFlow with varying state size.

duration for different state sizes grouped by the StateFlow component. We omit the 'execution graph traversal' component: this function is not split and therefore has no execution graph. As a first observation, we see that varying state sizes do not heavily impact any components but the 'state serialization'. Most operations are executed in ≤ 0.1 ms. The 'state serialization' component duration grows for increasing state sizes. This growth is an expected effect as serialization time is directly affected by the size of the serialized data. In essence, this graph shows the performance of the pickle serializer [4] used by StateFlow. Although not affected

[4] Pickle is a built-in serialization framework in Python. Besides Pickle, StateFlow also supports JSON and Protobuf serialization.

in this experiment, the same is true for event serialization. To improve this performance, developers can implement a more performant serialization framework.

Figure 6.7 shows the results for experiment with various **execution graph lengths**. Again, we use a logarithmic scale for the duration (y-axis). First of all, the length of this graph does not impact the performance of the 'routing', 'entity construction' and the 'state serialization'. These components still take ≤ 0.1 ms. Interaction with the execution graph is minimal for these components, and its size does not influence their performance. As expected, the traversal duration

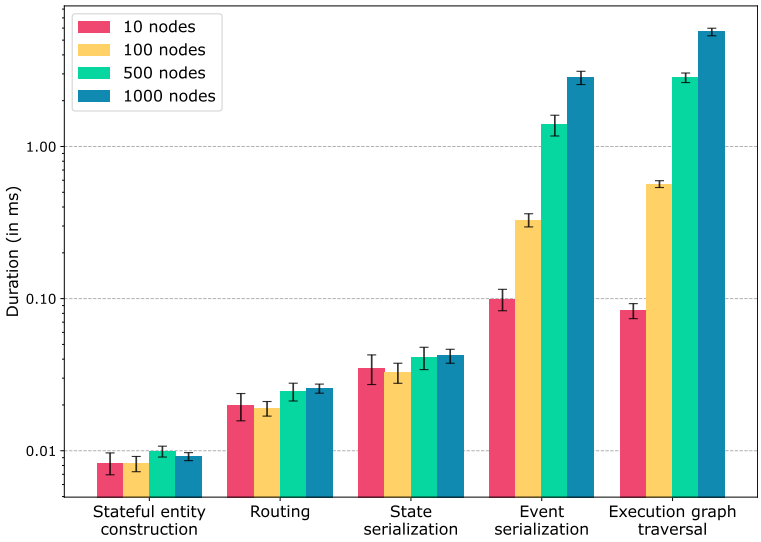


Figure 6.7: Duration of components in StateFlow with varying lengths for the execution graph.

increases as the size of the execution graph grows. Looking at the code example in Figure 6.4, the last statement returns variable x which has been defined in the first statement. In other words, the last node requires variable x, and StateFlow traverses the execution graph backward to find this declaration in the first node. As mentioned before, this is a naive approach and has a negative impact on performance. In the Chapter 7, we discuss potential improvements. Finally, the event serialization duration also increases as the size of the graph increases as Stateflow embeds the graph into the event.

Figure 6.8 shows the results for the experiment where a stateful entity has various **interactions**. This time, we show the durations on a linear scale (y-axis). We see a linear growth for all components as the number of interactions increases: as the number of interactions doubles, the duration also doubles. Invoking a method x times in isolation incurs the same overhead as if one stateful entity would invoke x other stateful entities plus the costs of execution graph traversal. To reduce overhead, StateFlow should improve the execution graph traversal performance and compress its size to improve event serialization.

6.3.2 Overhead *with* runtimes

Although the overhead experiments without runtimes indicate the performance of StateFlow, it does not give any perspective on the relativity of this overhead

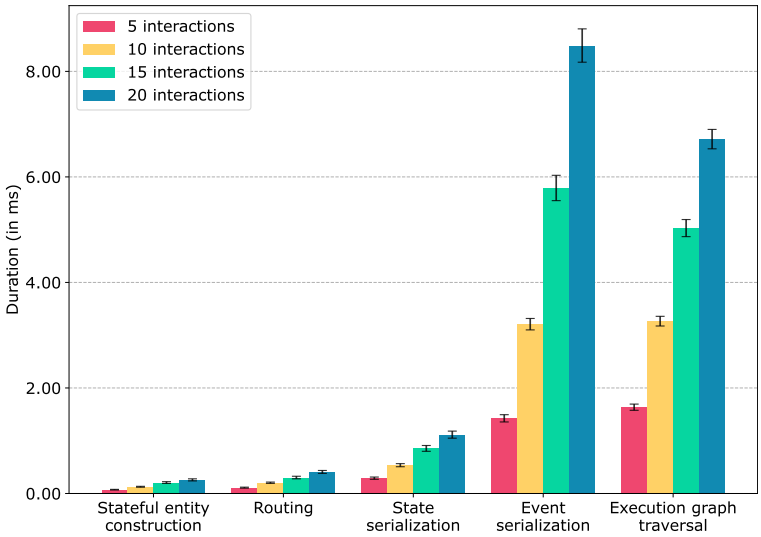


Figure 6.8: Duration of components in StateFlowwhen interacting with other stateful functions.

in the complete system. Therefore, we repeat the experiments from the previous section, but we also measure the overhead of the different runtimes. We consider all implemented runtimes except Apache Beam and CloudBurst: AWS Lambda, Flink Statefun, PyFlink, and Flink JVM. We exclude Apache Beam and CloudBurst because they both suffer significant bugs to be deployed outside a local IDE [5].

Experimental setup

Apart from AWS Lambda, we deploy all runtimes in a local setup using minimal resources. Most runtimes use batching mechanisms or parallel computation to improve performance. For example, Statefun sends events from the Flink cluster to the remote Python functions using batches. In addition, AWS Lambda automatically parallelizes the execution of incoming events, which might influence the performance of components like DynamoDB. As the goal of these experiments is to identify the overhead of single events, we try to avoid these mechanisms. We do this by sending events to the runtimes synchronously with intervals of 1 second to avoid batching. We have access to an Intel Core i5-6600k CPU @ 3.5GHz and 32GB of RAM on the local machine. A downside of these setups is that some require more resources than others. For example, the streamings systems require a local Kafka cluster, whereas the FaaS setup with AWS Lambda does not. As a result, StateFlow performance is slightly different for each setup. Therefore, we also time and report the StateFlow overhead again for these experiments.

AWS Lambda We configure AWS Lambda with 1024MB of memory and a max duration of 6000ms. We configure DynamoDB to be in on-demand mode, allowing it to scale with the number of reads and writes. Finally, AWS Lambda functions are invoked from the client using AWS Gateway. We do not include the overhead

[5] Beam suffers a bug with Kafka such that no records are received: https://issues.apache.org/jira/browse/BEAM-11998. Similarly, CloudBurst cannot be deployed as all Docker images are broken and out-of-date https://github.com/hydro-project/cloudburst/issues/62.

of AWS Gateway in this experiment as we focus on the overhead inside the runtimes. In AWS Lambda, we explicitly control key locking and interaction with state, as these are function calls in the Lambda handler. Therefore, we measure the overhead of these components individually. We consider reading and writing of state as two separate overhead components, whereas we combine the overhead of key locking and unlocking as key unlocking is often very fast.

Statefun, PyFlink and Flink JVM Setups for these three runtimes are relatively similar, and all rely on an Apache Flink cluster. We deploy a single JobManager and a single TaskManager with only one task slot. We configure both the Job and TaskManager to use 1GB of memory and have access 1 CPU core. We use such a minimal setup since we measure overhead for single events and do not require parallelism. Finally, we use a single-broker local Kafka setup to facilitate communication from the client to the runtimes. For Statefun, we also run a single-threaded web server in which we execute the remote Python functions. For the Flink JVM setup, we configure AWS Lambda with 1024MB of memory and a max duration of 6000ms.

For these runtimes, it is impossible to measure overhead for the stateful entity invocation steps as presented in Section 6.3. For example, reading and writing state is abstracted away in these runtimes, and without adjusting their internal code, it is impossible to measure the overhead of these operations individually. Moreover, due to the nature of streaming systems, there is no explicit locking of keys. Instead, events for keyed operators are executed sequentially for equal keys. This simulates the effect of key locking. Similarly, reads and writes to the state are abstracted away and cannot be measured individually. Therefore, we measure the overhead of the complete system rather than single components. A schematic overview for this approach can be found in Figure 6.9.

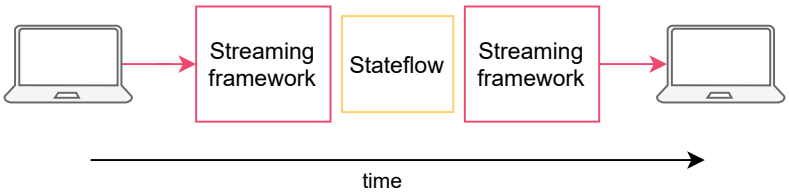


Figure 6.9: Schematic overview on how overhead in the runtimes PyFlink, Statefun and Flink JVM is computed.

To measure overhead, we attach a timestamp whenever the client sends an event to the streaming framework. Whenever this event arrives at the stateful operator where StateFlow handles the event, we compute the time elapsed using the previously attached timestamp. When StateFlow finishes processing the event, a new current timestamp is attached to the event. Finally, when the event arrives back at the client, the elapsed time is computed again. To compensate for communication from client to runtime (and the other way around), we deduct the Kafka latency twice. We consider this a constant latency and compute it by synchronously sending a message to Kafka and immediately reading that same message. We repeat this process 10000 times to get a realistic estimate. For a local Kafka setup, this resulted in single-trip latency of 7.18ms. In the case of

PyFlink and Flink JVM, events cycle through Kafka whenever a stateful entity interacts with another entity. We do not compensate for these interaction cycles. These cycles are a limitation of these runtimes and therefore count towards their overhead.

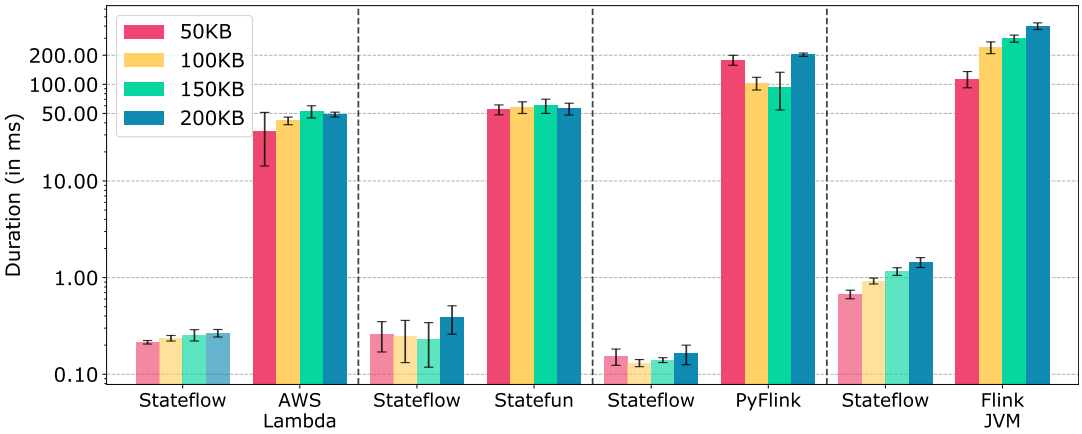
For these setups, we repeat the experiments from Subsection 6.3.1 presented in Table 6.3. Therefore we again consider three types of experiment: the overhead of having a stateful entity with various state sizes, execution graph lengths, and the number of interactions. The only difference is that for the state size experiment, we choose state sizes of 50KB, 100KB, 150KB, and 200KB. The reason to modify these sizes is because DynamoDB, part of the AWS Lambda setup, restricts the size of single data items. In the experiments *without* runtime we compute overhead of individual StateFlow components. For this experiments *with* runtimes, we consider the total overhead of StateFlow by summing the overhead of all these components. Contrary to the experiments without runtimes, we only repeat these experiments a 100 times. On the other hand, we do use the same class definitions for the stateful entities and similar client-side experimental code.

Results

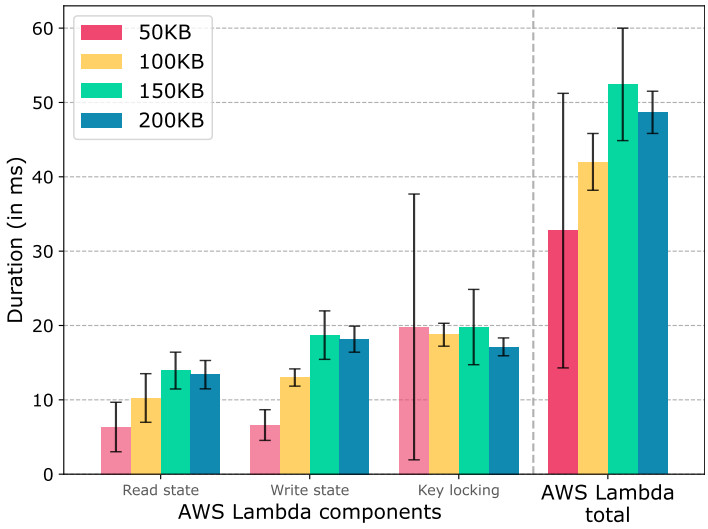
Figure 6.10 shows the results for the experiment with various **state sizes**. Figure 6.10a shows the overhead for each runtime, whereas Figure 6.11b shows the breakdown of each component in AWS Lambda. We do not have such a breakdown for the other runtimes as we can not compute overhead of its components individually. As mentioned before, depending on the setup, StateFlow performs slightly differently in each runtime. Therefore we report the overhead of StateFlow separately for each runtime. For Figure 6.10a, we use a logarithmic y-scale as the duration results have a wide range.

Figure 6.10a shows that all runtimes, apart from Flink JVM, are not hugely affected by increasing the size of the state. For the Flink JVM setup, we make two observations: 1) StateFlow's overhead is generally higher than StateFlow's overhead in other runtimes, and 2) we see a growth in StateFlow and Flink JVM durations as the state size increases. We attribute the first observation to the increase in event serialization costs. In the Flink JVM setup, we send the state and the event to the remote Lambda function. After computation in the remote AWS Lambda function, the updated state and event are sent back to the Flink cluster. This communication requires serialization and therefore increases these StateFlow costs. The second observation also relates to sending state to the remote AWS Lambda function. As the state increases, not only serialization but also communication costs increase. We attribute serialization costs to StateFlow and communication costs to the Flink JVM setup. Therefore overhead grows for both these components as the state size increases.

In the other runtimes, we do not see such effects. For the StateFlow component, the reason is that other runtimes make less use of event and state serialization. Each runtime uses the StateFlow building blocks differently, resulting in less or more overhead. For example, the Flink JVM uses event serialization twice to send events to AWS. Furthermore, we argue that, in this experimental setup, state sizes might be too small to see any significant effect. The results from the experiment without runtime (Figure 6.6), where we used larger state sizes, back up this hypothesis. In the Flink JVM setup, the overhead of increasing state



(a) Overhead for each runtime: AWS Lambda, Statefun, PyFlink and Flink JVM.



(b) Overhead breakdown for AWS Lambda.

Figure 6.10: Absolute duration of runtime overhead, including StateFlow, for varying state sizes.

size is simply amplified as it requires twice the amount of serialization and costly communication over the internet (i.e., from the local machine to AWS Lambda).

PyFlink and Flink JVM have the highest overhead of all runtimes. Not seen in this figure, but the most significant part of the overhead in Flink JVM runtime is communication to AWS. More specifically, $\leq 3\%$ is non-communication overhead. PyFlink is slower than most other runtimes because it is a new Python integration of Flink with several performance issues [6].

In Figure 6.10b we do see increase of state read and write overhead when the size increases, but this increase does not hold for all state sizes. Moreover, reading and writing state has similar duration and key locking is the most expensive operation. The latter involves multiple interactions with DynamoDB making it more complex than just reading and writing state.

To put all overhead into perspective, Table 6.4 shows the relative overhead of StateFlow in each runtime. It shows that for all state sizes, StateFlow contributes

[6]For example, in Flink version 1.13, Python operators are not chained: https://issues.apache.org/jira/browse/FLINK-23616

to less than $\leq 1\%$ of the total overhead.

	State size			
	50KB	**100KB**	**150KB**	**200KB**
Stateflow % of AWS Lambda	0.65%	0.62%	0.34%	0.54%
Stateflow % of Statefun	0.47%	0.42%	0.38%	0.68%
Stateflow % of PyFlink	0.09%	0.13%	0.15%	0.08%
Stateflow % of Flink JVM	0.58%	0.38%	0.39%	0.36%

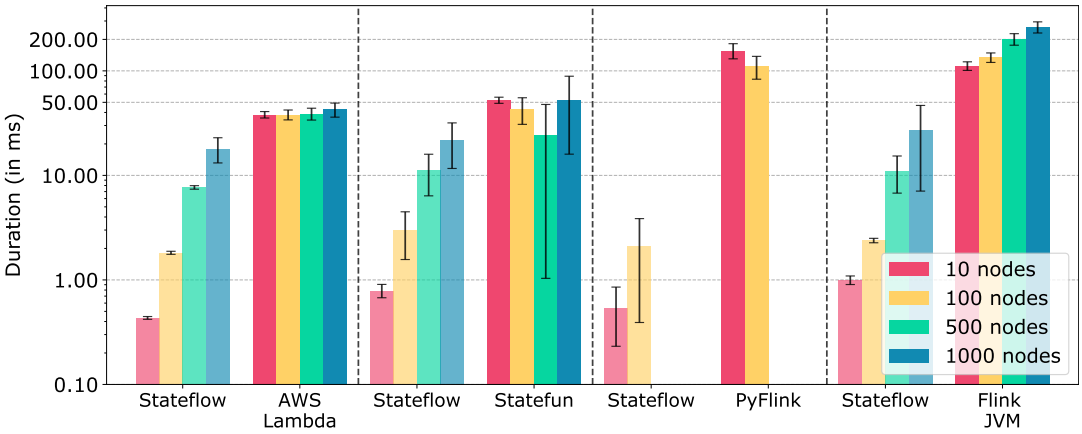
Table 6.4: Relative overhead of StateFlow in the different runtimes for various state sizes.

In Figure 6.11 we present the results for experiments where we vary with the **length of the execution graph**. Figure 6.11a shows the results for each runtime, whereas Figure 6.11b shows a breakdown for AWS Lambda. Once again, StateFlow is presented per runtime due to slight performance differences. For Figure 6.11a a logarithmic y-scale is used. Unfortunately, the experiment could not be executed with PyFlink for 500 and 1000 nodes. Due to a bug in PyFlink, classes with a lot of control flow elements resulted in a serialization exception. Therefore, these results are missing in the figure and table.

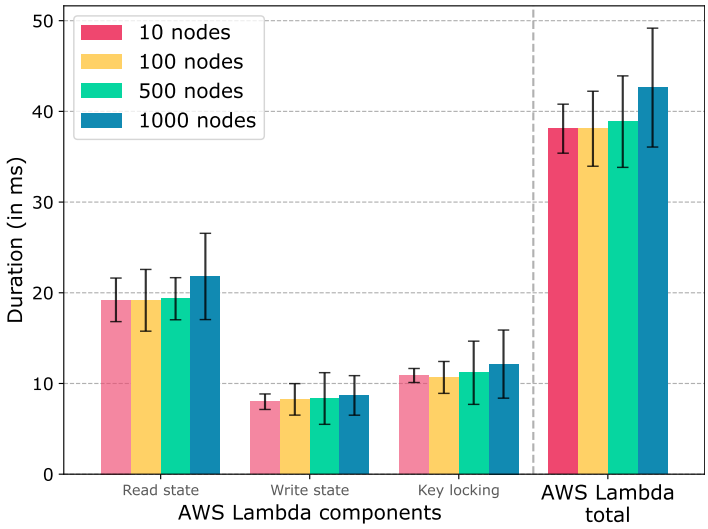
For the StateFlow component, regardless of the runtime, we see an increase in overhead as the execution graph length increases. This result is similar to that of the experiment without runtime (Figure 6.7). For the runtime overhead, we only see an increase in the Flink JVM setup. In this setup, the highest cost is sending events to and from AWS Lambda. We assume that a larger event size has a higher latency to AWS. StateFlow embeds the execution graph into an event, and therefore we can see this increase in the runtime overhead. All other runtimes are barely affected because their setup is different, similar to the state size experiment. In general, runtimes do not directly deal with the execution graph and are not affected by its length. If we look at the detailed overhead of AWS Lambda in Figure 6.11b, we see this confirmed. Finally, both PyFlink and Flink JVM have the highest overhead for the exact same reasons as mentioned in the state size experiment.

In Table 6.4, we see the relative overhead of StateFlow as part of the total overhead. As the execution graph increases, StateFlow plays a more significant role in the total overhead. For all configurations, StateFlow is responsible for $\leq 32\%$ of all overhead. However, these results do confirm that the execution graph approach and its implementation is one of the weak spots in StateFlow. We discuss this in more detail in Chapter 7.

Figure 6.12 shows the results for the experiments where we vary with the number of **interactions**. Figure 6.12a shows overhead of the runtimes, whereas Figure 6.12b shows a breakdown specifically for AWS Lambda. For each runtime the StateFlow performance is shown, due to performance differences. Finally, a logarithmic scale is used for the y-axis of Figure 6.12a.



(a) Overhead for each runtime: AWS Lambda, Statefun, PyFlink and Flink JVM.



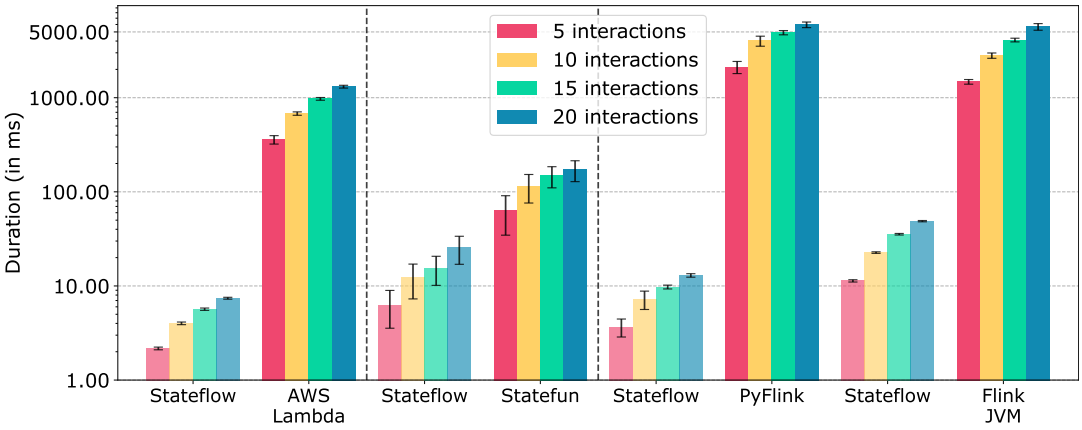
(b) Overhead breakdown for AWS Lambda.

Figure 6.11: Absolute duration of runtime overhead, including StateFlow, for various execution graph lengths.

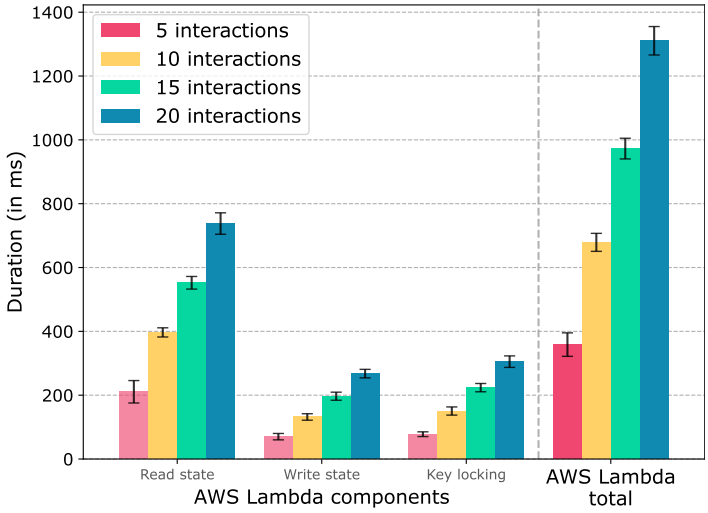
	Execution graph length			
	10	**100**	**500**	**1000**
Stateflow % of AWS Lambda	1.12%	4.55%	16.49%	29.74%
Stateflow % of Statefun	1.48%	6.57%	31.8%	29.29%
Stateflow % of PyFlink	0.35%	1.89%	-	-
Stateflow % of Flink JVM	0.89%	1.74%	5.19%	9.29%

Table 6.5: Relative overhead of StateFlow in the different runtimes for various execution graph lengths.

The main takeaway from this figure is that interactive stateful entities are expensive. The performance is in the order of seconds for most runtimes, whereas



(a) Overhead for each runtime: AWS Lambda, Statefun and PyFlink.



(b) Overhead breakdown for AWS Lambda.

Figure 6.12: Absolute duration of runtime overhead, including StateFlow, for varying amount of interactions **O2.7**-**2.9**.

this experiment still excludes end-to-end communication latency. State in streaming systems is partitioned, and for an entity to invoke another entity, an event needs to be re-routed to the corresponding (streaming) operator. In PyFlink and Flink JVM, we cycle to another operator in the streaming dataflow graph by ingesting the event back into this graph via a Kafka source. Repartitioning events is an expensive operation, mainly because streaming systems are optimized for acyclic graphs. Similarly, in the AWS Lambda runtime, we have to read, write and lock state from DynamoDB sequentially. Interestingly, Statefun is an order of magnitude faster than all other runtimes. It indicates that the internal routing system for Statefun, which StateFlow relies on, is much more optimized. PyFlink and Flink JVM have similar overheads, although there is no apparent reason for that. Again, the Flink JVM overhead mainly consists of communication latency to AWS. We believe that there is still much room for improvement in the runtimes and StateFlow. We discuss these extensively in Chapter 7.

Table 6.6 shows the relative overhead. For all runtimes but Statefun, StateFlow is responsible for $\leq 1\%$ of the overhead. With Statefun as a runtime, StateFlow takes up to $\approx 13\%$ of all overhead. StateFlow's share in the total overhead

is roughly the same for the first three configurations (i.e., 5, 10, and 15). This indicates that both Statefun and Stateflow have the same overhead growth for these parameters. However, this trend seems to break for 20 interactions.

	Number of interactions			
	5	**10**	**15**	**20**
Stateflow % of AWS Lambda	0.60%	0.59%	0.58%	0.56%
Stateflow % of Statefun	9.06%	9.62%	9.45%	12.90%
Stateflow % of PyFlink	0.17%	0.17%	0.20%	0.22%
Stateflow % of Flink JVM	0.76%	0.80%	0.85%	0.85%

Table 6.6: Relative overhead of StateFlow in the different runtimes for various interactions.

6.4 Performance

In the performance experiments, we evaluate StateFlow and its runtimes using the DeathStar benchmark (Section 6.1). Note that StateFlow is not a runtime system and only provides functionality at the event-level. In other words, StateFlow is not responsible for how the complete system scales and deals with increasing workloads. Therefore, these experiments are mainly to show how these runtime systems scale for general-purpose cloud applications. On the other hand, some design decisions in StateFlow, like its stateful entities, did influence the architectures of the underlying runtimes.

Using the DeathStar application, we design two types of experiments. First, a **low-throughput** experiment where we query each endpoint in isolation. We benchmark each endpoint for 30 seconds with 10 requests per second. To deal with performance variability, we repeat the experiment three times. In the second experiment, we query all endpoints and **gradually increase the throughput**. We experiment with 200, 300, 400, 500, 600, 700 and 800 requests per second for 60 seconds. We follow the DeathStar setup and distribute the requests. More specifically, 60% of the requests are to the */search* endpoint, 39% to the */recommend* endpoint, and only 0.5% to the */user* and */reserve* endpoint. This request distribution attempts to mimic a real-life scenario. To perform the benchmark, we rely on the wrk2 tool [7]. wrk2 is an HTTP benchmarking tool and produces a constant throughput load for a specific time. After benchmarking, it outputs a detailed report on the latency. For all experiments, we report end-to-end latency in milliseconds. We define this end-to-end latency as the time it takes to send the HTTP request and retrieve its result. For these experiments, we took inspiration from the evaluation done by the authors of the Beldi framework Zhang et al. [2020].

[7] https://github.com/giltene/wrk2

6.4.1 Experimental setup

In the setup, we distinguish between the frontend and the runtime. The setup for the frontend is equal for all experiments. However, the setup per runtime differs. Again, we exclude Apache Beam and CloudBurst as they suffer bugs preventing deployment. This time, we also exclude PyFlink from the increasing throughput experiment. We do include PyFlink for the low-throughput experiment. In preliminary performance experiments, PyFlink underperformed heavily, and most benchmark requests timed out. We dedicate this poor performance to the immaturity of the PyFlink framework. We believe that PyFlink's performance will be improved infuture releases. Therefore in these experiments, we include the runtimes: AWS Lambda, Flink Statefun, PyFlink (only low-throughput experiment) and Flink JVM.

We consider the frontend to consist of the benchmarking tool wrk2 and a set of HTTP servers. Figure 6.13 gives an overview. The wrk2 benchmark tool runs on a single AWS machine configured with 8 CPUs and 32GB of RAM. An AWS load balancer distributes requests over the different HTTP servers. Each HTTP instance serves the different endpoints in StateFlow's FastAPI integration. The HTTP instances relay the events to the runtime. Depending on the runtime, this is an event to Kafka topic or a direct request to AWS Lambda. These HTTP instances are stateless, and we scale them to 20 replicas. We configure each instance with 1 CPU and 1GB of RAM. To orchestrate these deployments, we rely on a Kubernetes cluster. We configured the frontend services with enough resources such that they never become the bottleneck. To prevent cold starts, we run a part of the workload before the experiments.

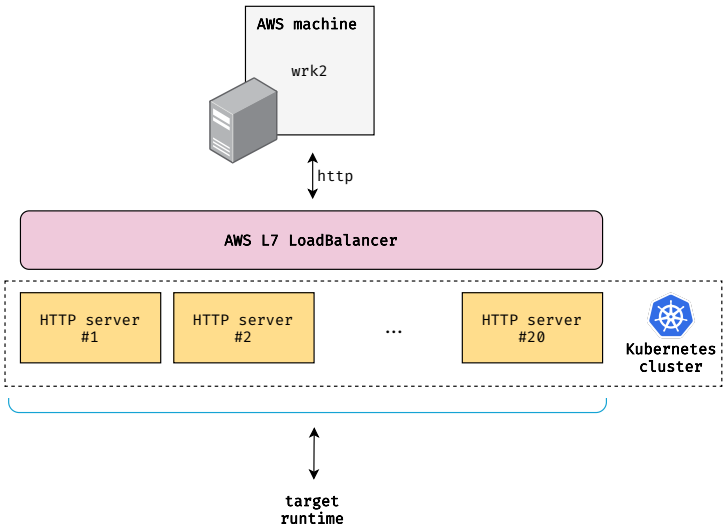


Figure 6.13: Generalized view of the 'frontend' architecture for the experiments.

AWS Lambda The setup for AWS Lambda is relatively simple. The HTTP instances directly invoke AWS Lambda. Amazon handles distribution over the different instances and auto-scales whenever necessary. We set the max concurrency of AWS Lambda at 1000. Moreover, we configured DynamoDB in on-demand

mode which ensures auto-scaling. Each Lambda instance has 1024MB and a max-duration of 6000ms. We disabled key-locking as preliminary experiments showed it had detrimental effects on the performance.

Flink Statefun Figure 6.14 visualizes the setup we use for the Statefun runtime. The frontend setup (Figure 6.13) is minimized to one block in this figure. The frontend communicates with the runtime via Kafka. We configure a Kafka cluster in Confluent Cloud [8]. We create all topics in the Kafka cluster with 40 partitions. Each TaskManager has access to 8GB of RAM and 2CPUs. For the Apache Flink cluster, we deploy 8 TaskManagers with five slots and set parallelism to 40. Finally, we deploy 20 HTTP servers for the remote Python execution. Each server is configured with 1GB of RAM and 1 CPU. Requests to these instances are load-balanced by an AWS load balancer. For all services, we use the same Kubernetes cluster as for the frontend. However, we force the Statefun remote servers and the Flink cluster to reside on different physical servers than the HTTP instances.

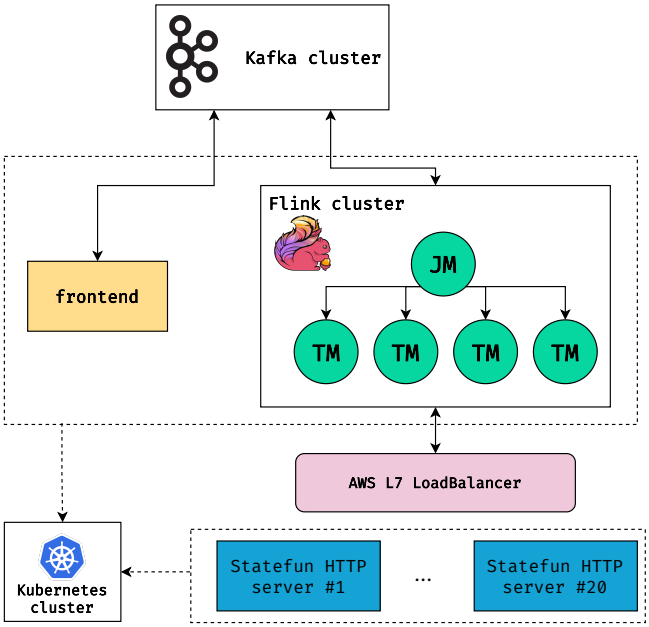


Figure 6.14: Experimental setup for StateFun and (partly for) PyFlink and Flink JVM.

PyFlink For PyFlink we use the exact same setup as for Statefun (Figure 6.14). However, PyFlink does not rely on remote Python functions and performs computation directly in its TaskManagers. Therefore, the load balancer and the HTTP servers are not part of PyFlink's setup.

Flink JVM Similar to Statefun and PyFlink, we use the Flink cluster for the Flink JVM setup. However, the Flink JVM setup outsources its computation to AWS Lambda. Looking at Figure 6.14, the TaskManagers send requests to AWS Lambda rather than local HTTP servers. We configure AWS Lambda with 1024MB of RAM, a max-concurrency of 1000, and a max-duration of 6000ms.

[8] Confluent offers Kafka as a cloud service: https://www.confluent.io/confluent-cloud/

6.4.2 Results

We show the results for the experiment with **low-throughput** and isolated endpoints in Figure 6.15. This figure shows the average end-to-end latency for each endpoint with ten requests per second. Note that a request to an endpoint might involve multiple calls to stateful entities and therefore the x-axis shows the amount of calls rather than the number of HTTP requests. For example, the */search* endpoint involves nine calls.

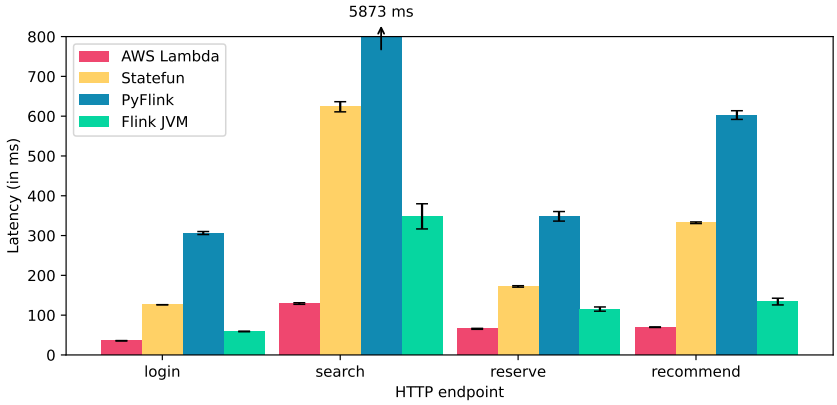


Figure 6.15: Average latency per DeathStar endpoint with 10rps.

The figure shows that AWS Lambda has the lowest latency for all the endpoints. Disabling key-locking improves the performance drastically. On the other hand, AWS Lambda does not give us any guarantees compared to the Flink setups. Whereas Statefun, PyFlink, and Flink JVM guarantee atomicity for single keys, AWS Lambda does not. The lack of atomicity is no issue for the read-only entities, but write entities might experience race conditions. In the hotel service scenario, the AWS Lambda setup might result in doubly booked rooms. Similarly, Flink setups provide us with an exactly-once guarantee. Such a guarantee does not exist for AWS Lambda. The performance differs the most for the */search* endpoint, and AWS Lambda outperforms the others by a large margin. The reason for that is that AWS Lambda handles split functions in a single Lambda invocation. Split functions require moving back and forth between operators at different (physical) locations in all other setups.

Interestingly, Flink JVM performs better than the Statefun runtime. In the overhead experiments, we saw Statefun outperform Flink JVM on all occasions. These overhead experiments also showed that the most significant overhead in the Flink JVM is the latency to AWS Lambda. In the performance experiments, however, the setup is much different. Flink JVM is deployed in an AWS environment rather than a local machine. Moreover, both AWS Lambda and all services are deployed in the same (physical) region. This setup reduces the inter-service latency and explains the improved performance of Flink JVM. Finally, PyFlink has the worst performance, especially for the */search* endpoint. Again, we attribute this to the immaturity of the PyFlink integration.

Figure 6.16 shows the results for the mixed workload with **gradually increasing throughput**. At the x-axis, we show the throughput as calls to stateful entities

rather than requests per second to the frontend. We argue that this gives a more realistic impression of the actual throughput. For example, 100 requests to the frontend result in 620 calls to stateful entities. We did not execute this experiment for PyFlink. The figure shows the average and 99th-percentile latency.

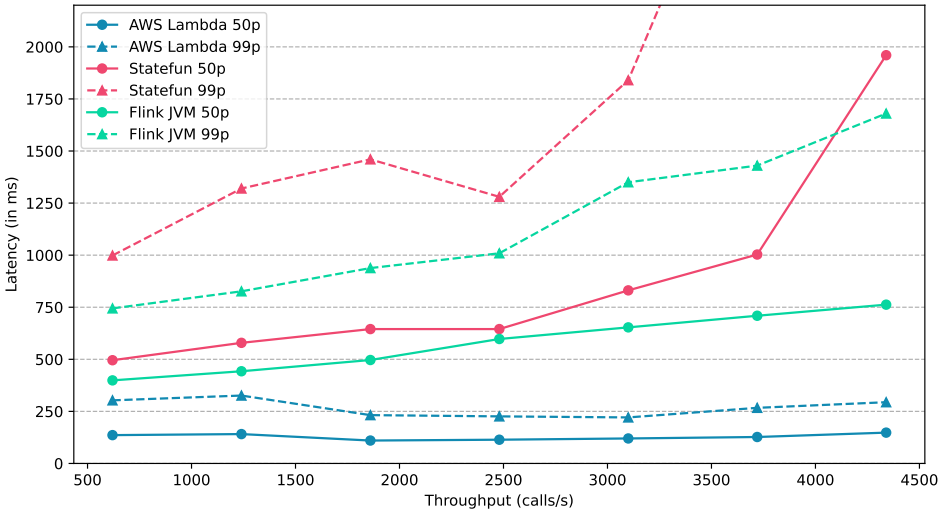


Figure 6.16: Average and 99th percentile latency for a mixed DeathStar workload with increasing throughput.

We conclude that AWS Lambda has the best performance, which does not degrade for an increased workload. With AWS Lambda autoscaling capabilities, we get resources up to 1TB of memory and around 580 CPUs[9]. At the same time, DynamoDB scales automatically with the amount of reads and writes. The resources for all other setups are much more scarce and static (i.e., no autoscaling). Therefore, one could argue that setups are not comparable in terms of resources. In addition, AWS Lambda's performance is at the cost of an exactly-once guarantee and atomicity for equal keys.

In the Statefun setup, we see results deteriorate after 3000 calls per second with a 99th percentile latency exceeding 2 seconds. For the Flink JVM setup, we see the latency increase more gradually with the increased throughput. However, we do not observe the same performance drops as in the Statefun setup. In the presented results, it is hard to argue why results deteriorate for specific runtimes. We argue that a more extensive evaluation is necessary to identify bottlenecks in each runtime. For example, an overhead evaluation for bigger workloads (i.e., higher throughput) should reveal potential shortcomings. At the same time, we believe StateFlow is not the cause of the lack of scalability as it only operates at the event level.

[9]In AWS Lambda, one configures the maximum RAM per instance, and AWS allocates CPUs proportional to the amount of RAM. According to AWS, an 1024MB instance has access to 0.58 CPUs. With a max-concurrency of 1000, this is equal to 580CPUs

Chapter 7

Discussion

In this chapter, we discuss the key findings, implications, and limitations of StateFlow. Furthermore, we present the related work. To structure the discussion, we divide this chapter into three parts. First, we discuss and interpret the evaluation results in-depth in Section 7.1. Second, we discuss the contributions of this thesis in Section 7.2. Finally, we compare StateFlow to existing work in Section 7.3.

7.1 Experimental results

In Chapter 6, we present a thorough evaluation of StateFlow. In this evaluation, we primarily focus on the expressiveness of the programming model, the overhead of StateFlow, and performance for runtime systems. We acknowledge that this evaluation does not cover all aspects of StateFlow. For example, we omit a performance analysis of the compiler pipeline. We argue that such an evaluation has a low priority, as the compiler only has one-time costs. In other words, the compiler only incurs initialization costs on the runtime and client systems. We consider these initialization costs negligible.

Similarly, we did not evaluate the client-side overhead in isolation similar to that of the runtimes. However, we know that the client-side overhead is minimal by design. StateFlow performs no computation at the client-side and only constructs, sends, and receives events. Therefore, we argue that such an evaluation does not offer valuable insights.

The rest of this section elaborates on the key findings from the conducted experiments.

Expressiveness In the expressiveness evaluation, we compare the DeathStar benchmark implementation in StateFlow to a native implementation in the supported runtime and the original Go implementation. We show that StateFlow's implementation requires the least amount of code and close to no non-application code. We conclude that StateFlow excels with its programming model in two ways: 1) developers only have to focus on application logic, while 2) developers still have control over operational aspects like the serialization framework. In addition, as StateFlow compiles code to an IR, switching runtimes does not require refactoring application logic. This runtime decoupling prevents a vendor lock-in not only with respect to cloud providers but also technologies like Apache Flink. It empowers developers and organizations to select a runtime that matches their features and even pricing preferences. If StateFlow does not support a preferred

runtime yet, integration requires little effort as all the building blocks already cover most functionality.

StateFlow requires developers to implement all application logic following the object-oriented paradigm. Additionally, all runtime code must be encapsulated in class definitions. One might argue this is a rather restrictive model, but we argue object-oriented code is an appropriate abstraction for large-scale applications with the potential to almost-infinitely scale. In this argument, we follow the reasoning of the work by Helland [2016]. Uniquely-identifiable entities are 1) an intuitive scale-agnostic programming abstraction for applications and 2) scale naturally by partitioning over multiple machines.

Since StateFlow decouples the programming interface and runtime system, one might miss out on runtime features. For example, Flink Statefun supports sending delayed events to stateful functions. StateFlow focuses on general-purpose large-scale applications, but some use cases require tailored solutions. In this instance, a developer might require a native implementation rather than using StateFlow. On the other hand, we envision that StateFlow supports more features embedded into the building blocks in the future. Features such as monitoring, logging, and even consistency models like multi-entity transactions.

Finally, StateFlow does not support all Python language constructs in the class definitions. As a result, developers cannot write arbitrary code. For example, the compiler pipeline does not support exception handlers (i.e., try/catch). In Section 4.5, we elaborate on all missing constructs. We argue that the existing support provides workarounds for unsupported constructs, and therefore, the programming model covers most implementations. Moreover, supporting more constructs is primarily an implementation effort, and we consider this future work.

Overhead In the overhead experiment, we evaluate StateFlow with and without runtime against experiments with different variables: various state sizes, various execution graph lengths, and different number of interactions.

Looking at StateFlow in isolation, it performs in the order of nanoseconds for routing and entity construction, reducing overhead to a minimum which is negligible in practice. The bottlenecks of StateFlow are its serialization and execution graph traversal which operate in the order of milliseconds. We use serialization for event communication and state storage, whereas the execution graph stores an execution plan for split functions.

Currently, the execution graph is a somewhat naive and unoptimized solution. We see multiple paths for improvement and envision this as future work. First, we argue for modifying the function splitting algorithm such that we only split (nested) control flow if it contains a remote call. Currently, we split for all control flow nodes (e.g., `if` or `for` AST nodes) to simplify implementation. Only splitting for remote calls reduces the number of nodes in the execution graph and optimizes its performance. A smaller execution graph also improves serialization performance as it reduces the size of the graph.

Moreover, we see room for optimization in the execution graph traversal approach. At the moment, we traverse the graph backward to find previously declared variables. This backward traversal is a greedy approach with a time complexity of $O(n)$ where n is the length of the execution graph. Instead, we envision having data dependency edges in the execution graph. These edges encode which

nodes have a data dependency and reduce the traversal to constant time $O(1)$. We can deduce these dependencies in the compiler pipeline by building a *definition-use* chain [1].

Another naive aspect of the execution graph is its storage of all usage and definitions variables. We do not analyze if subsequent nodes in the execution graph require any of the defined variables, and we simply store all. A *liveness analysis* in the compiler pipeline could identify unused variables in subsequent nodes and prevent storing them after function execution. Such an approach would optimize the execution graph by reducing its size and improving the serialization speed.

Finally, the execution graph is inherently sequential, and parallelization is not straightforward. For example, consider a function with a for-loop in which we call remote functions. In some instances, we could optimize this for-loop by executing each iteration in parallel. In theory, the dataflow model allows this kind of parallelization but is limited by the execution graph. To parallelize this execution graph, we require some sort of *synchronization* point. In terms of consistency and fault-tolerance, such synchronization is challenging to implement.

We argue that StateFlow has little control over the serialization overhead. In these experiments, we use Python's built-in *pickle* for both the event and state serialization and rely on its performance. To improve this performance, we recommend experimentation with other (faster) serialization frameworks like *serpyco* [2]. As StateFlow integrates with runtimes using building blocks, changing the serializer is trivial.

If we compare StateFlow's overhead to that of the runtimes, we see it is negligible for most configurations: StateFlow is only responsible for less than 1% of the total overhead. This minimal overhead shows that StateFlow does not sacrifice much performance in favor of its programming and execution model. The exception to this finding is the execution graph traversal experiment, but we see many potential improvements as mentioned above. Still, at its worst, StateFlow is only responsible for less than 32% of the total overhead.

Performance In the performance experiments, we benchmark the runtime systems against the DeathStar workload. We demonstrate how end-to-end latency changes as the throughput increases. In other words, we evaluate the *scalability* of the complete system. StateFlow operates at the event level and contributes a constant overhead. This overhead does not change as the system scales, and therefore, these experiments show the performance of underlying runtimes.

The DeathStar benchmark is a microservice benchmark and does not necessarily fit the concept of stateful entities (i.e., applications with partitioned state). For example, some components in the benchmark require access to the global state. Since dataflow systems partition their state across all operators, access to the global state is not trivial. As a solution, we use read-only replicas to distribute

[1] A definition-use chain is a data structure storing a definition and all usage variables reachable from that definition. This chain is a common technique used in static code analysis, used for example, for compiler optimizations Stoltz et al. [1994].

[2] There exist many Python serialization frameworks, this benchmark shows that serpco is among the fastest: `https://voidfiles.github.io/python-serialization-benchmark/`

the load. However, we argue that a benchmark with a stateful and partitioned application would have been a better fit. Unfortunately, to the best of our knowledge, no public benchmark exists at this moment.

The results of the performance experiment show that AWS Lambda has the lowest and most consistent latency, whereas Flink and Statefun show increased latency for higher throughput. Nevertheless, it is hard to directly compare runtime systems as each has its own features and characteristics. At the same time, it is hard to reason why performance deteriorates, and we argue that more extensive evaluation is necessary to identify bottlenecks. In particular, combining an overhead experiment and performance experiment might give fruitful insights. Although reasoning about specific bottlenecks is hard, we still present some tentative conclusions and recommendations regarding the runtime system based on our evaluation. Again, we divide the runtime systems into two categories: dataflow systems and (S)FaaS solutions.

Compared to AWS Lambda, dataflow setups like Apache Flink have a poorer performance. On the other hand, Flink offers guarantees like exactly-once and atomicity for single keys, which AWS does not. The benchmark results did not reflect such features. At the same time, the Python implementations of dataflow systems are immature and contain bugs. We expect that, over time, these Python API's will improve.

Furthermore, we argue that dataflow systems are a good fit for general-purpose cloud applications in terms of consistency guarantees, fault tolerance, and parallelization of computation. In terms of performance, there is still room for improvement, and we propose two specific research directions. First, we argue the **lack of querying the global state** in dataflow systems prevents the adoption of a more generalized workload. The lack of a global state forced us to change the DeathStar benchmark slightly and use read-only replicas. We envision that, at runtime, operators in dataflow systems should be able to query the global state such that the system behaves similarly to a database. We already see efforts made by frameworks such as Apache Flink by allowing state access from outside the compute cluster, but this is still premature and experimental. Second, we argue for **optimizing operator-to-operator communication** in dataflow systems. We require this communication between operators to support split functions. Currently, it requires roundtrips through external systems like Apache Kafka to make this work. Results in Figure 6.15 show that this approach is costly. Dataflow systems are built around directed acyclic execution graphs, and many features such as fault tolerance (i.e., checkpointing) rely on this principle. Therefore, adding cycles in the stateful dataflow graph is not straightforward. Nevertheless, we think an effort should be made to allow cycles, for example, by supporting forwarding events directly from a dataflow sink to a source rather than cycling through Kafka.

We observe the best overall performance for FaaS solutions like AWS Lambda (i.e., cloud functions), but at the cost of guarantees. These missing guarantees are not reflected in this benchmark. For example, AWS Lambda does not offer fault-tolerance nor delivery guarantees. The lack thereof might result in an unreliable application. Moreover, DynamoDB, which is part of the AWS Lambda setup, does not natively support key locking. As a result, execution for equal stateful entities is not atomic and might result in race conditions. We attempted to use an external library for key locking on DynamoDB, but preliminary experiments showed this

library does not scale.

7.2 Remaining contributions

In this section, we highlight all the contributions from this thesis and discuss the limitations and implications which are not covered by the evaluation results (Section 7.1).

Compiler pipeline In StateFlow's compiler pipeline, we transform object-oriented code to event-driven stateful dataflow graphs. The compiler analyzes each annotated class's AST and derives several static properties like method names, parameters, and instance variables. Moreover, we transform functions with remote invocations to a continuous-passing style form to support event-driven runtime architectures.

The major advantage of such a compiler pipeline is that developers write object-oriented Python code regardless of the underlying execution engine (i.e., runtime). Furthermore, it abstracts away from operational aspects, such as serialization, allowing developers to focus solely on business logic. Finally, developers enjoy the guarantees and features of a runtime without explicitly integrating these in the application code.

We reflect on two limitations of this compiler pipeline. First, as mentioned before, it restricts a developer in its programming model: it cannot compile arbitrary Python code. To resolve this, we envision the compiler to be extended, covering more Python constructs. Second, compiling code to a completely different representation and transforming code is not always transparent for a developer. As a result, reasoning about code and debugging code becomes more difficult. We try to improve this transparency by offering utilizations tools to visualize the compiled code. For example, we visualize the state machine of a split function which shows how the code is split (see Figure 4.17).

Finally, in this work, we do not formalize and prove any of the code transformations. Ideally, we prove that a split function has the same semantics as the original function. Instead, we use an empirical evaluation to assess its correctness. Formalizing and proving these transformations are not in the scope of this work, but we envision this as future work.

Stateful dataflow graphs In this work, we use stateful dataflow graphs as an intermediate representation for cloud applications. We propose a translation from object-oriented code to these graphs and how to execute these on top of several runtimes. Dataflow systems show that such a graph representation provides a perfect model for low-latency parallel computation. We now reflect on some of the shortcomings of stateful dataflow graphs as a representation for general-purpose applications. Most of these shortcomings, correspond to the discussed issues with the dataflow systems.

First of all, a stateful dataflow graph is less suitable for operator-to-operator communication (i.e., remote calls). One cannot directly invoke other operators and communication requires event messaging between them. To solve this, we compile functions with references to other functions in a continuation-passing style which abstracts this communication away from the developer. Moreover, we

provide execution graphs at runtime to coordinate the execution of such functions. In addition, a stateful dataflow graph representation is less suitable for having a global state. The graph assumes the state is partitioned across operators (i.e., nodes). We rely on underlying runtimes to support some sort of global querying.

Building blocks, integrations and deployment tools StateFlow provides building blocks to execute the stateful dataflow graphs on the different runtime systems. These building blocks have two major advantages. First, it decouples actual execution from the underlying runtime and gives StateFlow control over this logic. As a result, more functionality can be embedded into building blocks without touching integration code. Second, with the help of building blocks, code for integration with runtimes is minimal. Therefore, adding new runtimes is simple and integration with all supported runtimes required less than 190 lines of code. A disadvantage of using building blocks is the extra overhead. However, we show that for most situations, this overhead is minimal.

Besides building blocks to compose runtimes, StateFlow offers integrations with event streaming clients such as Kafka, an HTTP client, local unit tests, and tools for deployment. We argue that these integrations and tools offer an end-to-end 'experience' for the developer. In other words, StateFlow simplifies the implementation, testing, and deployment of scalable cloud applications.

Finally, we envision an ecosystem around StateFlow, supporting more runtimes, clients, and deployment tools. Such an ecosystem allows developers to simply write their application code and follows a plug-and-play approach to deploy this application in their desired setup.

7.3 Related work

In this section, we discuss work related to this thesis. In Subsection 7.3.1, we discuss related work on different paradigms for distributed programming and compare it to the paradigm proposed in this thesis. Subsection 7.3.2 discusses a new generation of distributed applications and how it relates to the execution of StateFlow applications. Finally, in Subsection 7.3.3 we discuss work on program synthesis and domain-specific languages and compare it to the techniques we used in StateFlow's compiler pipeline.

7.3.1 Distributed Programming

There exist two programming paradigms for the interaction of loosely coupled distributed systems: message passing and shared memory Kshemkalyani and Singhal [2011]. In the former paradigm, components of the distributed system pass around messages, whereas in the latter, components have access to shared memory for their communication and coordination.

In this thesis, we only work with systems following the message-passing paradigm, and therefore, in this section, we will not discuss any work on the shared memory paradigm. We first elaborate on the actor model as its concept is close to the *stateful entities* introduced in this thesis. Secondly, we discuss the dataflow model since we use this model for the execution of these entities. We discuss the most notable programming languages and frameworks that encapsulate these models.

The Actor Model

The actor model originates from the work by Hewitt et al. [1973], where the authors introduced actors as a primitive for concurrent computation. This model builds on previous models of computation such as the early version of the Smalltalk programming language Deutsch and Schiffman [1984] and the Simula language Nygaard and Dahl [1978]. Figure 7.1 presents a visualization of the actor model as discussed in Agha and Kim [1999]. Actors are isolated and autonomous objects which encapsulate data and methods. Each actor has its private local data (i.e., state), and the methods encode the behavior of this actor. This behavior includes creating new actors, sending messages to other actors, and modifying its local data. Actors are autonomous because they run in their own thread of control. They interact with other actors and external environments through message passing which is asynchronous. Each actor has a unique name to which messages can be directed (i.e., the actor address). The messages in the mailbox of an actor are executed one at a time. This ensures that the execution of methods in a single actor is atomic.

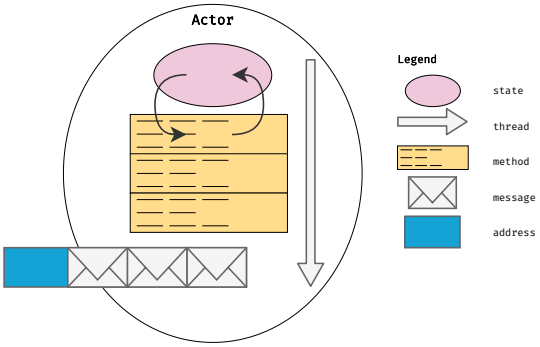


Figure 7.1: Visualization of the actor model. The figure is adopted from Agha and Kim [1999].

Looking at the figure, actors are very similar to the stateful entities described in this thesis. Stateful entities also encapsulate state and methods which are triggered through messages. Moreover, via the underlying runtime, we enforce single-thread execution and atomicity of method execution. However, we argue there are two main differences between the original actor model and StateFlow stateful entities. First of all, StateFlow entities are virtual and can 'live' everywhere as long as their state is available. Contrarily, actors are always 'alive' in their own control thread, waiting for the next message to arrive. The lifetime of a stateful entity is limited to its method execution. Bernstein et al. [2014] first introduced this idea as *virtual actors* in the **Orleans** framework. The second difference is that stateful entities abstract away from some constructs in the actor model. In other words, developers write object-oriented code and are not directly exposed to the concepts in the actor model. For example, to invoke a method on a stateful entity, one just calls the method rather than passing a message.

Programming Languages Some of the early actor programming languages include Rosette, Act 1, and Cantor proposed by respectively Tomlinson et al. [1989], Lieberman [1981], and Athas and Boden [1989]. However, most of these early languages were merely research prototypes and never had a significant adoption rate

in industry. However, the actor model has never been forgotten and it has been adopted in many established programming languages. We now discuss some of these languages.

The first 'industry-strength' language to adopt the actor model was **Erlang** Koster et al. [2016]. The Erlang language by Armstrong [2013] was initially developed within the telecommunications company Ericson in 1986 before it was open-sourced in 1998. In Erlang, actors are labeled *processes*, and everything within Erlang is a process. Each process is strongly isolated, and they only interact using message passing. One of the unique features of Erlang is that each process has its own memory (i.e., heap) which cannot be accessed by other processes and that garbage collection is optimized for this actor model.

The JVM language Scala also adopts the actor model by the name of **Scala Actors** introduced in Haller and Odersky [2009]. The authors claim Scala Actors to be a unification of thread-based and event-based actors. The developer can choose either for an actor to have its own thread (thread-based) and suspend this thread while waiting for a new message, or to use a thread-pool shared among multiple actors and only use a thread whenever a new message arrives (event-based). As opposed to Erlang, Scala Actors do not explicitly isolate the memory of each actor. On the other hand, Scala is more of a general-purpose language and offers other features besides its actors.

Many more modern programming languages are embracing the actor model, including **Dart**[3], **Scala** Odersky et al. [2004], and **Elixir** [4].

Frameworks Besides the programming languages encapsulating actors, there are also several concurrency frameworks embracing the actor model. Most notably, these frameworks support the execution of actors in a distributed setting. Therefore, these frameworks allow the scaling of actors beyond a single machine. Moreover, they offer more advanced features like transactions and fault tolerance.

Akka is an open-source JVM toolkit for distributed computing with an emphasis on actor-based concurrency Wyatt [2013]. Like Erlang, Akka implements most concepts in the actor model, such as a single thread and private state for each actor. Akka provides a distributed runtime allowing the actors to scale. Communication with actors in Akka is guaranteed to be at-most-once or at-least-once. Akka does not support the exactly-once guarantee. Each actor in Akka has a physical reference, and its location is fixed at creation. This physical actor location prevents dynamic load balancing, migration of actors, and machine failure handling Bernstein et al. [2014].

The work by Bernstein et al. [2014] and Bykov et al. [2011] introduced the **Orleans** framework. Orleans was the first to introduce virtual actors, which has been adopted in this thesis. Virtual actors are not bound to a physical location in the system and can always be 'activated' at any location. Like StateFlow, in Orleans, one can write object-oriented C# code where each object is an actor. Moreover, each actor in Orleans is also addressable by its type and unique key. In Orleans, these virtual actors are called grains. A silo, the term for a server instance, stores many of these grains. A new instance is created and stored in

[3] https://dart.dev/
[4] https://elixir-lang.org/

memory whenever a specific grain is invoked (i.e., activated). We show the complete lifetime of a grain in Figure 7.2. As Orleans persists grains, they can be moved between silos whenever necessary.

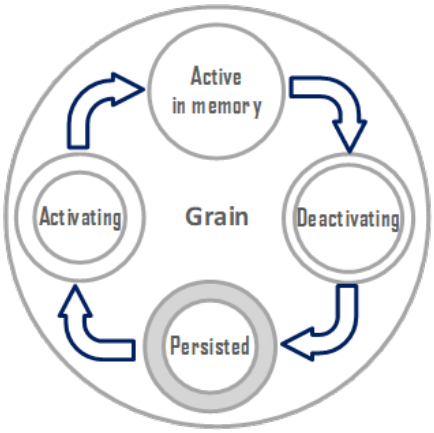


Figure 7.2: The lifetime of a grain in Orleans. Figure retrieved from the official Orleans documentation [5].

We identify two main differences between StateFlow and Orleans. First, StateFlow decouples state and compute. Unlike Orleans, we do not assign stateful entities to specific servers (i.e., Orleans silos). Stateful entities exist only in the underlying runtime's database or stateful operator, and its lifetime is limited to a method execution. Therefore StateFlow follows more of a serverless approach and integrates easier with the pay-as-you-go model. The second difference is that StateFlow abstracts away from the underlying runtime. Therefore, execution is not limited to a single distributed runtime, whereas Orleans tightly integrates its programming interface with its specialized runtime. In terms of message reliability, Orleans does not provide exactly-once guarantees. On the other hand, it does provide single-actor transactions. We do not have such guarantee in StateFlow.

The Dataflow Model

In the dataflow model, computation is data-centric, and one defines a program as a series of transformations and operations on this data. Often, the program is expressed as a directed graph in which nodes represent computation and edges represent how data flows from one transformation to another. Not only does the dataflow model refer to a language paradigm but also to a family of architectures based on this paradigm. In work by Whiting and Pascoe [1994], the evolution of historic dataflow languages and architectures are discussed.

The dataflow model gained popularity with the introduction of **MapReduce** by Dean and Ghemawat [2008]. MapReduce resulted in the Apache Hadoop framework, one of the most popular open-source large scale batch processing engines. In MapReduce, one defines a program as a *map* followed by a *reduce* operation. The underlying execution framework, like Apache Hadoop, takes care of the 'distributed' aspects like parallelization, communication, and fault-tolerance. The

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

work by Isard et al. [2007] introduced **Dryad** which was inspired by the MapReduce paradigm. Dryad extended the MapReduce concept by allowing user-defined operators defined in a directed acyclic graph (DAG).

Whereas MapReduce and Dryad still assumed data was provided in batches, many stream processing systems were introduced operating on unbounded data in the years following. For example, large companies like Linkedin, Google and Microsoft introduced respectively **Samza** Noghabi et al. [2017], **MillWheel** Akidau et al. [2013] and **Naiad** Murray et al. [2013]. At the same time, TU Berlin developed **Apache Flink** Carbone et al. [2015] and UC Berkeley **Spark Streaming** Zaharia et al. [2013].

Like StateFlow, popular streaming processing systems have recently been tested as a general-purpose execution engine for cloud applications. For example, Flink Stateful Functions demonstrates how the streaming engine Flink is leveraged for Function-as-a-Service applications. Interestingly, this thesis proposes to combine both the actor and the dataflow model—the actor model as a programming abstraction and the dataflow model as a way to execute these actors.

7.3.2 Stateful Functions

In recent years, a new breed of systems has arrived both from industry and academia. These systems are marketed as Stateful Functions or Stateful Function-as-a-Service (SFaaS) and include systems such as **CloudBurst** Sreekanti et al. [2020], **Apache Flink Statefun** de Heus et al. [2021], **Lightbend's CloudState** [6] as well as an early Scala prototype on top of Apache Flink Akhter et al. [2019]. Stateful Functions share similarties with the virtual actor model as discussed in Subsection 7.3.1. These systems often consist of a distributed runtime with features such as automatic scaling and fault tolerance. Moreover, often they include a more intuitive programming interface rather than low-level event handlers. For example, in CloudBurst, one can define and interact with Python functions which are then distributed and deployed onto a cluster. Opposed to StateFlow, each of these systems requires its specialized runtime. Besides, 'Stateful Functions' are often limited to single functions and do not support interactive entities as presented in this work. Moreover, due to the design of StateFlow and the use of building blocks, StateFlow allows for easy compiling to such distributed runtimes. In this work, we have shown this by supporting both Flink Statefun and CloudBurst as target systems with little integration code.

7.3.3 Program Synthesis

In this thesis, we present an approach to compile imperative Python code to a stateful dataflow graph. There are numerous works on translating imperative programs to other (intermediate) representations like dataflows or SQL. In this section, we highlight some of the works which embed a kind of program synthesis. Moreover, we discuss some work on continuation passing style as well as domain specific languages (DSL's).

With DBRidge Emani et al. [2017], developers write an imperative program that is translated to an SQL query. DBRidge detects parts of the code that can

[6] `cloudstate.io`

be expressed as SQL queries using static analysis. With the help of program transformations, DBRidge replaces these parts with SQL code. These analysis and synthesis techniques are similar to that of this thesis.

The work by Gupta and Sohi [2011] presents a model which executes sequential imperative programs on a multicore dataflow architecture. The proposed model dynamically extracts parallel tasks from imperative programs. Unlike our work, developers still have to specify which function invocations are part of the dataflow. Moreover, this thesis focuses on parallelizing and partitioning class instances rather than generic function invocations.

Ben-Nun et al. [2019] proposes a more generic synthesis technique. The authors present the Stateful DataFlow multiGraph (SDFG), an intermediate representation separating the program definition from its execution. Similar to StateFlow, one can define a Python program that translates to a dataflow graph. Unlike Stateflow, the work focuses on scientific code like matrix operations. Moreover the authors focused on optimizing their intermediate dataflow representation to improve the runtime performance. Such optimizations do not exist yet in StateFlow.

Gévay et al. [2021] presents a framework termed Mitos. Mitos abstracts away specific control flow elements, like while loops, into a dataflow graph. Mitos allows developers to write imperative programs (i.e., for loops, if statements) mixed with functional constructs (i.e., map, reduce). Moreover, it includes a runtime to execute these dataflow graphs. Unlike StateFlow, it does not support object-oriented code, nor does the intermediate dataflow graph compile to different runtimes.

In the field of machine learning, program synthesis is used to ease expressing complex machine learning programs. Similar to how StateFlow compiles imperative code to dataflow graphs, these works present approaches to compiling imperative code to an IR compatible with ML frameworks. For example, AutoGraph Moldovan et al. [2019] and Janus Jeong et al. [2019] convert imperative Python deep learning code into an intermediate graph representation. This IR can be executed on the TensorFlow framework [7].

Function splitting

This thesis proposes a 'function splitting' algorithm transforming imperative code to a continuation-passing style (CPS) Reynolds [1993]. Close to our approach is the work by Komondoor and Horwitz [2003b]. In this work, the authors propose automatic 'procedure extraction' aiming to simplify procedures (i.e., functions) by extracting some of the statements and replacing them with a procedure call. This way, the amount of code in the original function is reduced. Similar to StateFlow, functions are transformed at the syntax level while preserving the semantics of the original function definition. In follow-up work, this idea of procedure extraction is extended to eliminate code duplication Komondoor and Horwitz [2003a].

To compiler presented in Hemel and Visser [2011] for the Mobl language also applies transformations to ensure continuation-passing style. The Mobl language targets mobile web applications and integrates components, like user interface

[7] TensorFlow is one of the most popular deep learning frameworks: `https://www.tensorflow.org/`.

design and data modeling, into a single language. In these applications, some method calls, such as retrieving geolocations, are asynchronous. The compiler for Mobl transforms these asynchronous API calls to the continuation-passing style Hemel and Visser [2011]. As a result, developers can simply write synchronous code.

Domain Specific Languages

One aim of a domain specific language (DSL) is to abstract away from boilerplate code and therefore increase the productivity of a software developer Visser [2007b]. A good example is WebDSL Visser [2007a]; Hemel et al. [2008], which integrates several domains of web development, like data modeling and client-side interfaces, into a single language. Often a distinction is made between external and embedded domain-specific languages. An external DSL often has its own compiler or interpreter, whereas an embedded DSL is implemented within an existing host language. StateFlow falls in the latter category.

Chapter 8

Conclusion

This thesis presents StateFlow: a programming model, compiler pipeline, and execution model for general-purpose cloud applications. StateFlow compiles object-oriented Python code to a stateful dataflow graph and executes these on dataflow systems and (Stateful) Function-as-a-Service solutions. To conclude this work, we explicitly answer the research questions as presented in the introduction.

StateFlow allows developers to write object-oriented applications in which they do not have to consider operational nor infrastructural aspects. At the same time, we argue that stateful dataflow graphs are a proper intermediate representation for large-scale (cloud) applications. Therefore, we propose to compile object-oriented code to such an IR, by which we answer the first research question:

RQ1: How does one transform object-oriented code to event-driven stateful dataflow graphs?
In Chapter 4, we introduce the compiler pipeline of StateFlow. This pipeline compiles class definitions to stateful dataflow graphs through static analysis and code transformations. More specifically, in Section 4.1 and 4.2, we show how StateFlow analyzes the AST of classes in a Python program to derive static properties, such as method names and instance variables. Then, StateFlow builds a stateful dataflow graph from all analyzed classes. In this graph, each operator represents a class and stores the state of objects. To invoke a method of an object, a client sends an event into the dataflow graph with the call arguments. The result of the invocation flows out of the dataflow graph in the form of another event. Finally, StateFlow transforms the functions with remote calls into a continuation-passing style to make it compatible with the event-driven stateful dataflow graph, as explained in Section 4.3.

The compiler pipeline of StateFlow ends in a stateful dataflow graph for all analyzed classes. In turn, StateFlow integrates this with several distributed processing engines, by which we answer the second research question:

RQ2: Given a stateful dataflow graph, how does one execute this graph with loose coupling to an underlying distributed processing engine? In Chapter 5, we present StateFlow's building blocks: operators and routers. Together, these building blocks offer a comprehensive model of execution for stateful dataflow graphs. StateFlow handles all execution

logic in its building blocks regardless of the underlying runtime, so the execution is loosely coupled. In addition, integration with these runtimes is lightweight, and switching runtimes does not require any application code refactoring. We show the effectiveness of using these building blocks by supporting several types of architectures in less than 190 lines of code.

The use of an intermediate representation for general-purpose cloud applications allows us to benchmark and compare several runtimes executing identical application code. In this thesis, we primarily focused on dataflow systems and (S)FaaS solutions. This leads us to answering third and final research question:

RQ3: What is the performance and overhead of dataflow and (S)FaaS systems for general-purpose cloud applications, and what are the limiting factors? In Chapter 6, we experiment with runtime systems and StateFlow measuring both overhead and performance. Typically, the overhead added by the building blocks of StateFlow is minimal, incurring less than 1% of the total overhead. We observe that AWS Lambda, a FaaS solution, incurs the least overhead and has the best performance. At the same time, AWS Lambda has the *least guarantees*, which makes it prone to faulty and unreliable execution. For dataflow systems, such as Apache Flink, we conclude that performance is mainly limited by *expensive operator-to-operator communication* and a *lack of global state*. On the other hand, its exactly-once guarantee makes it a reliable runtime. At this point in time, selecting a runtime system is a trade-off between consistency guarantees and performance.

8.1 Future work

In the discussion, we already hinted at some future research directions for StateFlow. In this section, we elaborate several of those directions.

Extended Python support We argue that a successful embedded DSL does not significantly restrict the developer. A developer should be able to use almost all programming constructs of the underlying language. StateFlow currently covers around 70% of all Python constructs inside the class definitions. For example, one cannot use a try/catch statement or a list comprehension expression. Future work could extend StateFlow's programming model to fully support all Python constructs. In addition, StateFlow can be extended to support object-oriented concepts such as polymorphism and inheritance. Some of these extensions are merely an implementation effort, whereas others might require more tailored solutions.

Transactions Currently, some of the runtime systems provide atomicity for single-key stateful entities. This prevents race conditions when modifying the state of a single entity. However, StateFlow does not have such a guarantee for a function involving multiple entities. In that scenario, StateFlow moves back and forth

between operators, and parallel events might update the state in between, resulting in unexpected results. We consider transactions for multiple entities to be a promising research direction. We envision embedding such a guarantee in the building blocks of StateFlow, by relying on primitives provided by the runtime. From a developers perspective, one could then enable a transactional guarantee by annotating the particular method with `@transaction`. Again, StateFlow would abstract away the complexity of transactions and enforce transactional semantics at the runtime level. A starting point for this research direction is the work by de Heus et al. [2021], which explores distributed transactions for (S)FaaS.

Extend the support and optimize architectures Our work currently supports numerous runtime systems, including AWS Lambda, Apache Flink, and Flink Statefun. One of the strengths of this work is the ability to easily integrate the IR with new runtimes. Besides adding more runtimes, we envision adding different types of runtimes. This thesis focuses on the dataflow system and (S)FaaS solutions, but we argue StateFlow is not limited to these distributed runtime types. For example, distributed actor runtimes such as Akka or Dapr could be exciting additions [1], because each of these has a different set of guarantees, features, and performance and therefore are a better fit for specific applications. At the same time, we acknowledge the need for a more extensive evaluation of underlying runtimes. We envision that StateFlow has its own benchmark allowing a fine-grained evaluations of the performance and the overhead of the underlying runtime. Not only does this highlight potential bottlenecks, but it also enables the assessment of new runtimes.

Formal verification of the compiler We believe that a form of formal verification for the compiler would benefit the pipeline. It is essential that all code transformations implemented in the compiler pipeline preserve the semantics of the original function definition. In this work, we verified this through an empirical evaluation by implementing a benchmark. Formal verification of all proposed AST transformations would underline the theoretical correctness of StateFlow's compiler pipeline. To prove semantic preservation of AST transformations, it requires a formal definition of the Python semantics. Several works have already proposed formal operational semantics for Python Köhl [2021]; Smeding [2009]; Politz et al. [2013].

[1] `https://akka.io/` and `https://dapr.io/`

Bibliography

G. A. Agha and W. Kim. Actors: A unifying model for parallel and distributed computing. *J. Syst. Archit.*, 45(15):1263–1277, 1999. doi: 10.1016/S1383-7621(98)00067-8. URL https://doi.org/10.1016/S1383-7621(98)00067-8.

A. Akhter, M. Fragkoulis, and A. Katsifodimos. Stateful functions as a service in action. In *VLDB*, 2019.

T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, 2013. doi: 10.14778/2536222.2536229. URL http://www.vldb.org/pvldb/vol6/p1033-akidau.pdf.

T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015.

M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *SIGMOD*, 2018.

J. Armstrong. *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf, 2013.

W. C. Athas and N. J. Boden. Cantor: an actor programming system for scientific computing. *ACM SIGPLAN Notices*, 24(4):66–68, 1989. doi: 10.1145/67387.67402. URL https://doi.org/10.1145/67387.67402.

T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler. Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures. In M. Taufer, P. Balaji, and A. J. Peña, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, pages 81:1–81:14. ACM, 2019. doi: 10.1145/3295500.3356173. URL https://doi.org/10.1145/3295500.3356173.

P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. 2014.

S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *SoCC*, 2011.

P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink tm : Stream and batch processing in a single engine. In *IEEE Data Eng. Bull.*, 2015.

P. C. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. The rise of serverless computing. *Commun. ACM*, 62(12):44–54, 2019. doi: 10.1145/3368454. URL https://doi.org/10.1145/3368454.

A. Cheung, N. Crooks, J. M. Hellerstein, and M. Milano. New directions in cloud programming. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. URL http://cidrdb.org/cidr2021/papers/cidr2021_paper16.pdf.

B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010. doi: 10.1145/1807128.1807152. URL https://doi.org/10.1145/1807128.1807152.

M. de Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos. Distributed transactions on serverless stateful functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, DEBS '21, page 31–42, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385558. doi: 10.1145/3465480.3466920. URL https://doi.org/10.1145/3465480.3466920.

J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Communications of the ACM*, 2008.

L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *SIGACT-SIGPLAN*, 1984.

K. V. Emani, T. Deshpande, K. Ramachandra, and S. Sudarshan. Dbridge: Translating imperative code to sql. In *SIGMOD*, 2017.

Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 3–18. ACM, 2019. doi: 10.1145/3297858.3304013. URL https://doi.org/10.1145/3297858.3304013.

C. Gencer, M. Topolnik, V. Durina, E. Demirci, E. B. Kahveci, A. G. O. Lukás, J. Bartók, G. Gierlach, F. Hartman, U. Yilmaz, M. Dogan, M. Mandouh, M. Fragkoulis, and A. Katsifodimos. Hazelcast jet: Low-latency stream processing at the 99.99th percentile. In *VLDB*, 2021.

G. E. Gévay, T. Rabl, S. Breß, L. Madai-Tahy, J. Quiané-Ruiz, and V. Markl. Efficient control flow in dataflow systems: When ease-of-use meets high performance. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 1428–1439. IEEE, 2021. doi: 10.1109/ICDE51399.2021.00127. URL https://doi.org/10.1109/ICDE51399.2021.00127.

G. Gupta and G. S. Sohi. Dataflow execution of sequential imperative programs on multicore architectures. In *MICRO*, 2011.

P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009. doi: 10.1016/j.tcs.2008.09.019. URL https://doi.org/10.1016/j.tcs.2008.09.019.

P. Helland. Life beyond distributed transactions: an apostate's opinion. In *ACMQueue*, 2016.

J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019a. URL http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf.

J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019b. URL http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf.

Z. Hemel and E. Visser. Declaratively programming the mobile web with Mobl. In *OOPSLA*, 2011.

Z. Hemel, L. C. L. Kats, and E. Visser. Code generation by model transformation. In *Theory and Practice of Model Transformations, First International Conference, ICMT*, 2008.

C. Hewitt, P. B. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In N. J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245. William Kaufmann, 1973. URL http://ijcai.org/Proceedings/73/Papers/027B.pdf.

M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In P. Ferreira, T. R. Gross, and L. Veiga, editors, *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, pages 59–72. ACM, 2007. doi: 10.1145/1272996.1273005. URL https://doi.org/10.1145/1272996.1273005.

E. Jeong, S. Cho, G. Yu, J. S. Jeong, D. Shin, and B. Chun. JANUS: fast and flexible deep learning via symbolic graph execution of imperative programs. In J. R.

Lorch and M. Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 453–468. USENIX Association, 2019. URL https://www.usenix.org/conference/nsdi19/presentation/jeong.

E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, pages 445–451. ACM, 2017. doi: 10.1145/3127479.3128601. URL https://doi.org/10.1145/3127479.3128601.

R. Komondoor and S. Horwitz. Eliminating duplication in source code via procedure extraction. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2003a.

R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, pages 33–43. IEEE Computer Society, 2003b. doi: 10.1109/WPC.2003.1199187. URL https://doi.org/10.1109/WPC.2003.1199187.

J. D. Koster, T. V. Cutsem, and W. D. Meuter. 43 years of actors: a taxonomy of actor models and their key properties. In S. Clebsch, T. Desell, P. Haller, and A. Ricci, editors, *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016*, pages 31–40. ACM, 2016. doi: 10.1145/3001886.3001890. URL https://doi.org/10.1145/3001886.3001890.

A. D. Kshemkalyani and M. Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2011.

M. A. Köhl. An executable structural operational formal semantics for python, 2021.

H. Lieberman. A preview of act 1. 1981.

D. Moldovan, J. M. Decker, F. Wang, A. A. Johnson, B. K. Lee, Z. Nado, D. Sculley, T. Rompf, and A. B. Wiltschko. Autograph: Imperative-style coding with graph-based performance. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org, 2019. URL https://proceedings.mlsys.org/book/272.pdf.

D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In M. Kaminsky and M. Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 439–455. ACM, 2013. doi: 10.1145/2517349.2522738. URL https://doi.org/10.1145/2517349.2522738.

S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: Stateful scalable stream processing at linkedin. *Proc. VLDB Endow.*, 10(12):1634–1645, Aug. 2017. ISSN 2150-8097. doi: 10.14778/3137765.3137770. URL https://doi.org/10.14778/3137765.3137770.

K. Nygaard and O. Dahl. The development of the SIMULA languages. In R. L. Wexelblat, editor, *History of Programming Languages, from the ACM SIGPLAN History of Programming Languages Conference, June 1-3, 1978, Los Angeles, California, USA*, pages 439–480. Academic Press / ACM, 1978. doi: 10.1145/800025.1198392. URL https://doi.org/10.1145/800025.1198392.

M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. 2004.

J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The full monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, page 217–232, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323741. doi: 10.1145/2509136.2509536. URL https://doi.org/10.1145/2509136.2509536.

F. Raab. TPC-C - the standard benchmark for online transaction processing (OLTP). In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, 1972.

J. C. Reynolds. The discoveries of continuations. *LISP Symb. Comput.*, 6(3-4): 233–248, 1993.

J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson. What serverless computing is and should become: the next phase of cloud computing. *Commun. ACM*, 64(5):76–84, 2021. doi: 10.1145/3406011. URL https://doi.org/10.1145/3406011.

P. Silvestre, M. Fragkoulis, D. Spinellis, and A. Katsifodimos. Clonos: Consistent causal recovery for highly-available streaming dataflows. In *SIGMOD*, 2021.

G. J. Smeding. An executable operational semantics for python. *Universiteit Utrecht*, 2009.

V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(11):2438–2452, 2020. URL http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf.

Stoltz, Gerlek, and Wolfe. Extended ssa with factored use-def chains to support optimization and parallelism. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, volume 2, pages 43–52, 1994. doi: 10.1109/HICSS.1994.323280.

A. Subasi. Chapter 1 - introduction. In A. Subasi, editor, *Practical Machine Learning for Data Analysis Using Python*, pages 1–26. Academic Press, 2020. ISBN 978-0-12-821379-7. doi: https://doi.org/10.1016/B978-0-12-821379-7.

00001-1. URL https://www.sciencedirect.com/science/article/pii/B9780128213797000011.

C. Tomlinson, W. Kim, M. Scheevel, V. Singh, B. Will, and G. Agha. Rosette: An object-oriented concurrent systems architecture. *ACM SIGPLAN Notices*, 24 (4):91–93, 1989. doi: 10.1145/67387.67410. URL https://doi.org/10.1145/67387.67410.

E. Visser. WebDSL: A case study in domain-specific language engineering. In *GTTSE*, 2007a.

E. Visser. Domain-specific language engineering. In *Pre-Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*. Braga, Portugal, 2007b.

P. G. Whiting and R. S. V. Pascoe. A history of data-flow languages. *IEEE Ann. Hist. Comput.*, 16(4):38–59, 1994. doi: 10.1109/85.329757. URL https://doi.org/10.1109/85.329757.

C. Wu, J. M. Faleiro, Y. Lin, and J. M. Hellerstein. Anna: A KVS for any scale. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 401–412. IEEE Computer Society, 2018. doi: 10.1109/ICDE.2018.00044. URL https://doi.org/10.1109/ICDE.2018.00044.

D. Wyatt. *Akka concurrency*. Artima Incorporation, 2013.

M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 423–438, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522737. URL https://doi.org/10.1145/2517349.2522737.

H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 1187–1204. USENIX Association, 2020. URL https://www.usenix.org/conference/osdi20/presentation/zhang-haoran.