Cloud Monads: A novel concept for monadic abstraction over state in serverless cloud applications

Master's Thesis in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Jelle Eysbach
4480708

18th May 2022

Author
Jelle Eysbach

Title
Cloud Monads: A novel concept for monadic abstraction over state in serverless cloud applications

MSc presentation
May 27th, 2022

Graduation Committee

dr. L. Chen	Delft University of Technology
dr. J. S. Rellermeyer	Delft University of Technology
dr. B. Ahrens	Delft University of Technology

Abstract

Serverless computing is a relatively recent paradigm that promises fine-grained billing and ease-of-use by abstracting away cloud infrastructure for developers. There is an increasing interest in using the serverless paradigm to execute data analysis tasks. Serverless functions often interact with external services, which can be considered similar to the concept of side-effects in regular programming. Haskell uses monads to isolate side-effects and to structure the composition of functions using side-effects. This thesis explores whether the concept of the monad can be applied in a serverless computing environment, ideally in a way that is flexible with regards to platform. An abstraction of side-effects was developed in the form of a monadic layer that is added to serverless functions. The monadic layer interacts with monads using an interface and exposes the API of the monads to the user. A monadic implementation was also created for a platform-independent function composition mechanism using orchestrator functions. An implementation of a shared state side-effect has also been created as a practical use-case for the monadic layer, and to explore the usability of monads on platforms with more restricted composition frameworks. The implementations are evaluated on performance, expressiveness and usability.

Preface

From an early age I have been very interested in computers. At the time they seemed magical and my young self would not have expected that one day I would have the knowledge about them that I do now, having achieved an MSc. degree in computer science. I feel truly privileged to be able to understand and reason about the amazing progress that the field of computer science has been going through the past decades, and I look forward to seeing what is next.

Firstly, I would like to thank prof. Rellermeyer for the guidance and supervision during my thesis project.
I am also very grateful towards UbiOps for allowing me to work on my thesis with them, especially towards Victor and Kees, who have given me a lot of feedback and helped me stay on track during my time there.
I would also like to thank the other employees of UbiOps for providing a nice working environment and for their enthusiasm regarding my project.

Finally, I would like to thank my family and friends for always supporting me during my studies.

Jelle Eysbach

Delft, The Netherlands
18th May 2022

Contents

Chapter 1

Introduction

Serverless computing is a relatively recent paradigm that promises fine-grained billing and ease-of-use by abstracting away cloud infrastructure for developers. Instead of renting virtual machines (VM), developers simply supply the code that should be executed on certain triggers and the cloud provider will execute it on one of its machines when the trigger fires. This paradigm is incredibly well-suited to applications like stateless APIs due to their event-based and stateless nature, but there is an increasing interest in using the serverless paradigm to execute data analysis tasks [1, 2, 3, 4]. These tasks benefit greatly from the inherent scalability and low maintenance costs that serverless provides, and since these tasks may not be executing 24/7 the more fine-grained billing has the potential to reduce costs depending on the workload. However, the short lifespan of serverless functions require a data analysis task to be implemented as a workflow containing multiple separate functions. This causes the implementation of long-running analysis tasks to require some coordination that is generally not inherently available in serverless environments.

These long-running workflows would ideally be modelled as a composition of smaller functions that dictates how the data flows between functions and how the desired result is found. In such pipelines, the storage of intermediate data that is not fit to be sent through HTTP request has to be explicitly managed by using a storage solution outside of the serverless platform. The coordination of such pipelines is implemented differently on different platforms. For example, Fission and AWS Lambda rely on workflows defined in .YAML files[1][5, 6], while Azure Durable Functions allows the definition of an "orchestrator" as a serverless function [7].

Composed functions are limited by the fact that all serverless functions are restricted to being stateless. The short-lived nature of the containers causes there to be no guarantee what the state of any invocation's local storage will be. However, this construction is also what enables the scalability and elasticity of the server-

[1]YAML is a human-readable data-serialization language.

less paradigm. The only way that serverless functions can become stateful is by interacting with external services over a network. These interactions with external sources are very similar to the concept of side-effects in regular programming. In general programming, developers should try to isolate side-effects outside of business logic as much as possible because not doing so can make programs more difficult to test and debug. In serverless there is a case to be made for introducing side-effects in a controlled way to increase the expressiveness of the serverless functions. By providing certain side-effects as a platform, it is possible to limit the potential errors that users may make in an attempt to implement side-effects themselves. It will also allow developers to focus on their business logic, without having to contaminate their code with logic for handling side-effects. More complex concepts like consistency properties of state storage can be a source of issues for the unprepared but can be handled in platform-provided side-effect implementations.

The inspiration for this work is the concept that Haskell uses to isolate side-effects; monads. Monads are used to isolate the implementation of side-effects from pure, business logic functions. The monadic idea of separating side-effects from business logic and composing a sequence of monadic functions in a well-defined way would also be valuable in a serverless environment. This thesis will explore the ways in which this might be implemented.

1.1 Research questions

RQ1: Would monads offer an effective model of abstraction to simplify the implementation or use of side-effects in a serverless computing environment?

RQ2:Would monads be able to provide a standardised solution for serverless function composition?
The composability of monadic functions is one of the key features of monads. Therefore the composition of serverless functions should also be considered when exploring a monadic abstraction of side-effects.

RQ3: Can monads be used in serverless to model and control side-effects?
Monadic functions signal via their monadic return type that they use side-effects. A similar system may also be useful for serverless platforms to predict certain behaviours of a function and implement optimisations.

RQ4: What infrastructure would be required to enable cloud providers to provide users with side-effects on their platform?

1.2 Thesis structure

Chapter 2 explores the background of serverless computing, functional programming, Haskell specifically and finally contains a small note about Aspect-Oriented Programming. Chapter 3 summarises the different serverless function composition models that exist for the different serverless function offerings. In chapter 4, the similarities between functional programming and serverless computing are explained. It also provides details about how some common monads are implemented in Haskell. Chapter 5 explains the side-effect abstraction that was developed and chapter 6 details a shared state side-effect implementation for UbiOps. The following chapter 7 contains the evaluations that were performed on the developed work. Finally, in chapter 8 the results are discussed, as well as related work and future research.

Chapter 2

Background

This section provides an overview of the current state of the serverless environment, the properties of functional programming and introduces Haskell and monads.

2.1 Function-as-a-Service

Serverless computing, or Function-as-a-Service (FaaS), is a relatively recent addition to the cloud computing domain and is currently provided by each of the big three cloud computing providers; Amazon Web Services (AWS Lambda), Microsoft Azure (Azure Functions) and Google Cloud (Google Cloud Functions). FaaS allows users to deploy small functions to the cloud, which can be called via an API. The serverless platform takes care of all aspects of the deployment of the code. The cloud platform also automatically provides functions with a high degree of elasticity under different loads.

Before FaaS the predominant method of deploying applications in the cloud was using Infrastructure-as-a-Service (IaaS). Infrastructure-as-a-Service allows customers to rent Virtual Machines (VMs), virtual shares of computers which users need to install their own operating system and software stack on to. IaaS allowed companies to deploy their applications without having to rely on on-premise hosting and all the hardware, network and security management that it implies. IaaS also made it easier to scale up or down the amount of infrastructure being rented.
IaaS has some properties that make it inefficient for certain use-cases. E.g. VMs run continuously, meaning the customer pays for an idling machine if there are no requests coming in at a certain time. It is possible to implement elasticity but this requires more effort when compared to the serverless paradigm, which provides it by default.
Serverless functions provide a much finer granularity of compute resources. AWS Lambda, for example, scales CPU resources with memory and allows users to configure the amount of memory between 128 MB and 10,240 MB, with a 1,769 MB

function costing the equivalent of one vCPU credit [8]. In contrast, the smallest machine type of AWS EC2 costs 3 vCPU credits [9].
VMs also take a very long time to start up compared to serverless functions. [10] measured an average of 96.9 seconds for a Linux VM on EC2 as one of the faster configuration options, so it is not feasible to scale a service down to zero machines until a request comes in.

FaaS promises a number of benefits over IaaS. The first is the concept of abstracting architecture away from users and reducing the complexity of deploying code significantly. Users do not have to worry about managing VMs, transferring their code, monitoring the status of the VMs or configuring virtual networks. All users need to do is supply the code snippet for a function to the cloud platform and it will immediately and always be accessible. Another advantage is the inherent, automatic elasticity of functions. Serverless functions can be scaled up more easily, because containers can be started more quickly [11]. Functions are automatically scaled down to zero running instances, meaning the user does not get billed for idling functions.

FaaS also comes with some limitations. The functions are stateless unless an external service is used to store state. Local file storage can not be used between function invocations because there is no guarantee that a new invocation operates in the same container as previous invocations. Latency may also be a issue due to the cold start problem, meaning the latency of functions scaled down to zero containers is significantly increased compared to functions that already have a container running. However, the startup time of containers is still significantly lower than the startup times of VMs. The resources that are available to a serverless function are limited. They are generally limited in properties like memory allocation, CPU speed and function timeouts [12, 13, 14]. Each serverless platform is built up and interacted with in a different way. Existing code bases will have to be changed to function in a serverless setting [15]. Even between serverless platforms there are differences making migration difficult, meaning there will also be some degree of vendor lock-in.

The serverless paradigm is not inherently linked to a certain architecture but usually contain at least a user-facing component, a proxy to a function container and the function containers themselves. Current serverless offerings generally use a container management platform like Kubernetes and a container engine like Docker to host and execute functions. The controller is a user-facing component that the user interacts with via RESTful APIs. In OpenWhisk, the different components communicate via a distributed message queue like Kafka. The components of Fission communicate directly. Fission contains a component that keeps track of running containers. It is responsible for starting new function containers and communicating their IP addresses to the controller or router. This is called the executor in Fission. OpenWhisk does not keep function containers running, every invocation starts a new container and destroys it afterwards. Fission also contains a separate component that starts up builders for environments that require compilation or downloading of dependencies before creating a function pod image.

Functions are generally activated by triggers. For example, Fission supports four kinds of triggers: [16]

- HTTP Triggers: invokes a function on an HTTP request.
- Timer Trigger: invokes a function based on a timer.
- Message Queue Trigger: for invoking functions using message queues.
- Kubernetes Watch Triggers to invoke functions when something in your cluster changes

OpenWhisk provides actions, triggers and rules to interact with the platform [17]. Actions are the functions, triggers are events that may trigger execution of an action. Rules define the actions that should be executed on triggers.

The use-cases that fit FaaS the most naturally execute short functions as reactions to events [15]. Image processing is one example of such a class of functions [18]. As a workload, image processing scales very well due to the independent nature of the operation. Generating thumbnails or running object detection are some practical examples of image processing use cases.
Another example of a well-suited use-case is a web service back-end with a Representational State Transfer (REST) application programming interface (API). Interactions with a REST API are stateless so they fit serverless functions well. While it is relatively straightforward to implement new projects on serverless, it is difficult to port existing projects to a serverless platform [15].
Internet-of-Things (IoT) applications also fit serverless environments very well conceptually [19]. Sensor data measurements can be modeled as events and processed as a serverless function. This also moves data processing away from the sensors to the cloud, meaning low-power devices only have to send a simple HTTP request without the need for extra processing.

However, the serverless platform's fine-grained billing and elasticity properties would also be valuable in other applications, like long-running data analytics or machine learning jobs. These applications currently do not fit serverless platforms as well due to their longer run-time, intermediate storage requirements and synchronisation requirements. Intermediate data storage is a problem in serverless because using data locality is not possible in a serverless architecture. The data will always have to move to the code because the ephemeral serverless function containers can not be used to store data. Any data that is too big to fit in an HTTP request will have to be written to an external data storage, and read by the next function, which will have negative consequences on the runtime of the functions.

There are extensions to the serverless platform like AWS Step Functions and Azure Durable Functions that focus on function composition and orchestration. These enable longer run times through composition but do not solve the intermediate storage requirements [5, 7]. AWS Step Functions also uses a separate application to create the workflows, while it is arguably more convenient to be able to incorporate this functionality directly in the source code, like in Azure Durable Functions.

Due to the stateless nature of serverless and the focus on breaking software up in small re-usable functions, there are some conceptual similarities to functional programming with pure functions.

2.2 Functional programming

Functional programming, as opposed to imperative or object-oriented programming, takes a more mathematical approach to programming. Imperative programming is largely based on the underlying hardware of computing systems, with memory allocations and explicit thread management, also known as the Von Neumann model. Functional programming revolves around creating programs using a collection of functions following the mathematical definition of a function, meaning they map an input domain to an output domain.

2.2.1 Pure functions and side-effects

In programming, a function that follows the mathematical definition is called a "pure" function. Pure functions take their input as arguments, and do not produce any observable effect besides returning a result. Their output is also deterministic with regards to their input. Effects that violate one of these properties are called side-effects.
Listing 2.1 contains an example of a function with a side-effect. The function *incrementX* modifies the global variable *x*, meaning it has an observable effect besides returning a value.

```javascript
var x = 1;

function incrementX() {
    x += 1;
    return x;
}
```

Listing 2.1: Example of a JavaScript function with a side-effect

This example obviously causes a side-effect, but let us make *x* a parameter of *incrementX*. The presence of a side-effect now depends on whether *x* is passed as a value or as a reference, making it much less obvious. The mutation of input variables is another source of side-effects, and the reason why functional programming languages generally use immutable data structures.

Another example of a side-effect is a file system or database interaction. Pure functions are deterministic with regards to their input, but reading a file at different times may give different results. [20]

2.2.2 Properties

Functional programming restricts the use of mutable data structures and does not contain control flow constructs like loops. Recursion and higher-order functions should be used for iteration instead. There are a number of advantages that these restrictions and functional programming's approach to side-effects enable over imperative programming. The first advantage is that the runtime can reason about order of execution because all dependencies are made explicit through function arguments. The runtime can determine when the result of a certain computation is required, and defer execution otherwise. Another advantage is that functions are easier to reason about, and test, in isolation. There is no external state that the function could depend on, which could influence its execution. The main disadvantages of functional programming are the steep learning curve and the high memory use due to data immutability.

Pure functions have the aforementioned beneficial properties but programs ultimately need side-effects to interact with the world. The functional programming language Haskell ran into this problem during its design but has developed one of the cleanest solutions to separating side-effects from application logic in functional programming; monads.

2.3 Haskell and monads

There was a consensus at the "Functional Programming Languages and Computer Architecture" conference in 1987 that widespread adoption of functional programming languages in practice was limited by the fact that there were multiple similar languages. The proposed solution was to start a committee that would design a common functional language. This committee created Haskell. One of the stronger requirements for Haskell was that it should be a pure language. This requirement eventually led to the invention of monadic programming. [21]

A monad is an abstraction that allows the implementation of side-effects without breaking the pure functional nature of the functions [22]. A monad is essentially a wrapper around the return value of a function that can be composed with other function invocations in a sequence by using the bind (>>=) operator. Monads allow the implementation of side-effects to be abstracted away from business logic and allow side-effects to be represented in a pure way. They also provided a way for code to have a well-defined execution order. An example of a situation where this is required is printing something to the console. Printing to the console

does not return anything, so without monads Haskell will lazily never evaluate it because its result is never used.
A more in-depth explanation of monads can be found in section 4

Chapter 3

Serverless function composition models

The most interesting feature of monadic programming is the ability to compose multiple monadic functions in a sequence, without the user having to explicitly handle monadic values. Similarly, an abstraction of side-effects in serverless computing should also provide functionality for function composition. For example, it might be valuable for a parallel set of functions, or function invocations, to have access to a shared monadic environment. Unfortunately, each serverless platform has its own extension for function composition, which causes problems for users like vendor lock-in and makes porting functions to another platform or language more difficult.

An effective abstraction of side-effects should be applicable on different composition models, and functions using it should require minimal effort to port to other models. For this reason, this section explores the different implementations of serverless function composition. The effects of the differences between the composition frameworks on the architecture of the side-effect abstraction is explained in section 5.4

Function composition extensions can be categorised into three categories based on the way they model compositions: declarative, imperative, or functional. The categorization is displayed in Table 3.1

Declarative	**Imperative**	**Functional**
Fission Workflows	Google Cloud Workflows	OpenWhisk Composer
AWS Step Functions	Azure Durable Functions	FaaS-Flow sync chain
UBiOps pipelines		
FaaS-Flow async chain		

Table 3.1: Categorization of composition frameworks

3.1 Declarative composition model

The declarative composition models allow the user to define a Directed Acyclic Graph (DAG) or a finite state machine for their compositions. DAGs are commonly used as a representation of a workflow for task scheduling in distributed systems, for example in spark [23], High Performance Computing (HPC) architectures [24] and grid computing [25].

A declarative composition is defined by a number of steps that represent individual functions or operations. The steps are linked to each other by inter-dependencies or by explicitly referencing the following step to execute. The platform is responsible for creating an execution plan, executing steps in parallel where possible.

3.1.1 Fission workflows

Fission workflows are defined as a DAG using YAML files. Listing 3.1 shows the definition of a simple workflow. The workflow consists of two tasks, one of which has a dependency. The input of the tasks are defined with the help of a special JavaScript data expressions. The $ variable contains the complete state of the workflow invocation. In the example, the output of the workflow is defined on the second line of the composition. The scheduler component creates an execution plan based on the current state of an invocation.
Fission workflows also contains internal functions that are executed within the workflow engine itself and provide users with the possibility to implement control-flow constructs like conditional execution and loops. The internal functions can be called the same way as other fission functions and are represented as a single task in the DAG.

```yaml
apiVersion: 1
output: WhaleWithFortune
tasks:
  GenerateFortune:
    run: fortune
    inputs: "{$.Invocation.Inputs.default}"

  WhaleWithFortune:
    run: whalesay
    inputs: "{$.Tasks.GenerateFortune.Output}"
    requires:
    - GenerateFortune
```

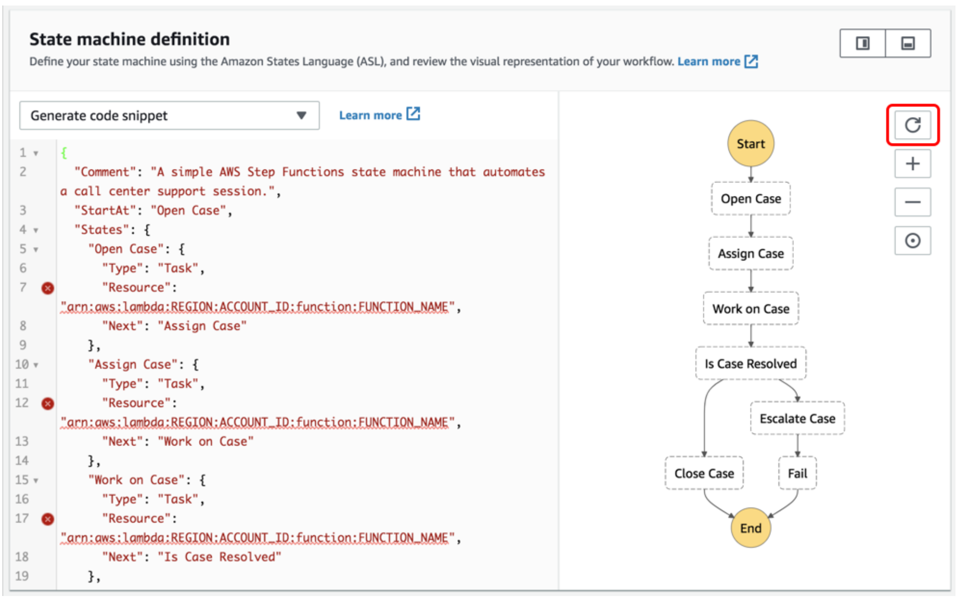
Listing 3.1: Example fission workflow definition. Source: https://github.com/fission/fission-workflows

3.1.2 AWS Step Functions

AWS step functions works with state machines instead of DAGs. They can be defined using the JSON-based, Amazon States Language (ASL) or using a graph-

ical user interface. Tasks are modelled as states in the state machine. Instead of defining dependencies and using expressions to link inputs and outputs, each task state contains a reference to the state that should be executed next. The outputs of a task are automatically used as input for the next task. An example of a workflow is shown in Figure 3.1.
AWS Step functions has different types of states that implement different functionality. For example, the *Task* state type executes a single unit of work, like an AWS Lambda function. Other state types like *Choice* and *Parallel* implement the control-flow constructs conditional branching and parallel execution.

Conditional branching is a feature that is not available in the DAG-based approach of Fission workflows. However, the DAG-based approach is able to automatically infer when parallelization is possible.


State machine definition

Define your state machine using the Amazon States Language (ASL), and review the visual representation of your workflow. Learn more

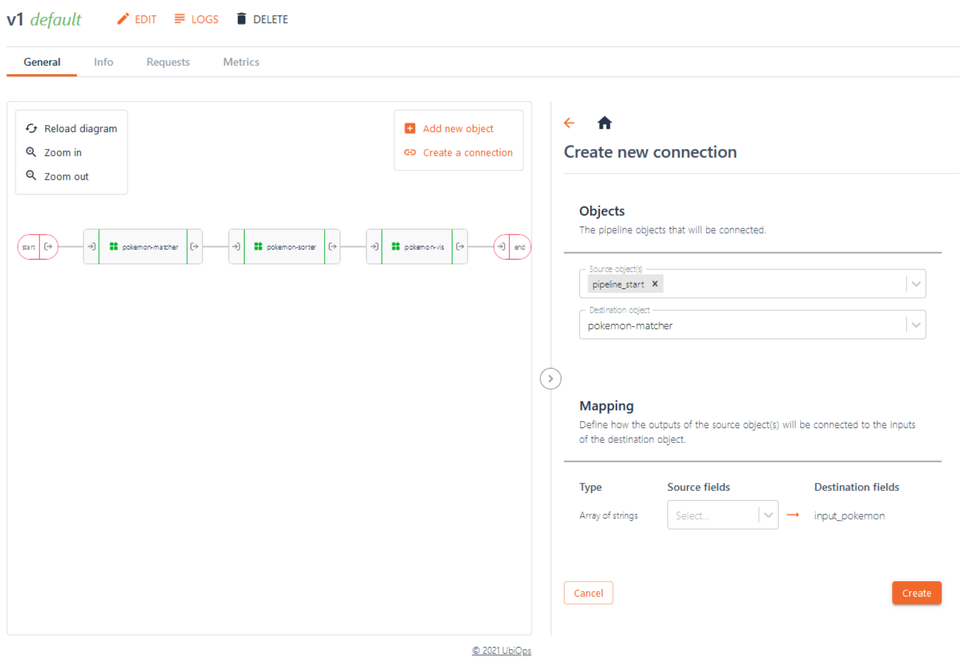
Generate code snippet ▼ Learn more

```
{
  "Comment": "A simple AWS Step Functions state machine that automates
a call center support session.",
  "StartAt": "Open Case",
  "States": {
    "Open Case": {
      "Type": "Task",
      "Resource":
"arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
      "Next": "Assign Case"
    },
    "Assign Case": {
      "Type": "Task",
      "Resource":
"arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
      "Next": "Work on Case"
    },
    "Work on Case": {
      "Type": "Task",
      "Resource":
"arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME",
      "Next": "Is Case Resolved"
    },
```

Figure 3.1: Example AWS Step Functions state machine definition. Source: https://aws.amazon.com/getting-started/hands-on/create-a-serverless-workflow-step-functions-lambda/

3.1.3 UbiOps pipelines

UbiOps pipelines can be defined with a graphical user interface, similar to AWS step functions, or with a YAML definition. The graphical user interface can be seen in Figure 3.2. Pipelines consist of functions and connections. A connection contains a mapping from a function's output to the next function's input. UbiOps functions' in- and outputs contain type definitions so the mapping is checked for type correctness. UbiOps pipelines only support sequences with diverging branches. If a pipeline diverges, the result of the pipeline is an array, where each element corresponds to the result of a branch.



Create new connection

Objects
The pipeline objects that will be connected.

Source object(s): pipeline_start
Destination object: pokemon-matcher

Mapping
Define how the outputs of the source object(s) will be connected to the inputs of the destination object.

Type	Source fields	Destination fields
Array of strings	Select...	input_pokemon

Cancel
Create

Figure 3.2: Example UbiOps pipeline. Source: https://app.ubiops.com

3.1.4 FaaS-flow

FaaS-flow is the composition extension for OpenFaaS. FaaS-flows are defined using Go, and have a distinction between sync chains and async chains. Sync chains more closely resemble the functional programming paradigm and will be covered in section 3.3. Although the definitions of async chain DAGs are created imperatively instead of using a format like JSON or YAML, the DAGs themselves stay a declarative way to model composition. The user does not have to implement any control-flow constructs or scheduling.
Listing 3.2 shows an example of a simple DAG definition that executes a sequence of three tasks. The snippet creates a DAG, adds nodes with functions to it, and finally creates the edges between the nodes. The snippet also contains a call to the method Modify for node n2, which allows using in-line Go functions to operate on the data.

```go
func Define(
    flow *faasflow.Workflow,
    context *faasflow.Context
) (err error) {
    dag := flow.Dag()
    dag.Node("n1").Apply("func1")
    dag.Node("n2")
        .Apply("func2")
        .Modify(func(data []byte) ([]byte, error) {
            // do something
            return data, nil
        })
    dag.Node("n3").Apply("func4")
    dag.Edge("n1", "n2")
    dag.Edge("n2", "n3")
    return nil
}
```

Listing 3.2: Example FaaS-flow DAG definition. Source: https://github.com/s8sg/faas-flow

3.2 Imperative composition model

Imperative composition models serverless function execution as a function call and allows composed functions to call other functions as such.

3.2.1 Azure Durable Functions

Azure durable functions is an example of this approach. Durable functions allows the definition of orchestrators, which are serverless functions themselves and can call activities using the method callActivityAsync. The orchestrator does not wait for the activity to finish but shuts down and is restarted when the activity is finished. Orchestrators can be written in C#, Python or JavaScript. In C# orchestrators are async functions and use the await keyword to signify that the result of the activity is required at that point. An Example of an orchestrator written in C# can be seen in Listing 3.3.

```
[FunctionName("FanOutFanIn")]
public static async Task Run(
    [OrchestrationTrigger] IDurableOrchestrationContext context)
{
    var parallelTasks = new List<Task<int>>();

    // Get a list of N work items to process in parallel.
    object[] workBatch =
        await context.CallActivityAsync<object[]>("F1", null);
    for (int i = 0; i < workBatch.Length; i++)
    {
        Task<int> task = context.CallActivityAsync<int>("F2",
            workBatch[i]);
        parallelTasks.Add(task);
    }

    await Task.WhenAll(parallelTasks);

    // Aggregate all N outputs and send the result to F3.
    int sum = parallelTasks.Sum(t => t.Result);
    await context.CallActivityAsync("F3", sum);
}
```

Listing 3.3: Example orchestrator function in C#. Source: https://docs.microsoft.com/nl-nl/azure/azure-functions/durable/durable-functions-overview

3.2.2 Google Cloud Workflows

Google cloud workflows defines a composition as a sequence of steps that can be written in YAML. It is very similar to the declarative approach of AWS step functions but it has features that more closely resemble imperative programming. The first of these features is that the order of the steps in the YAML implies execution order. This execution order can be overruled with jumps, using the next keyword to define the next step that should be executed. Google cloud workflows also allows users to define conditionals, which can be combined with jumps to enable different execution branches. Finally it is also possible to assign variables. Assigning a step's result to a variable is also the only way to access the results of that step in another step. Listing 3.4 contains an example of a google cloud workflow. The workflow performs a get request to callA and uses the result in the following conditional to call one of SmallFunc, MediumFunc and LargeFunc.

```
- first_step:
    call: http.get
    args:
      url: https://www.example.com/callA
    result: first_result
- where_to_jump:
    switch:
      - condition: ${first_result.body.SomeField < 10}
        next: small
      - condition: ${first_result.body.SomeField < 100}
        next: medium
    next: large
- small:
    call: http.get
    args:
      url: https://www.example.com/SmallFunc
    next: end
- medium:
    call: http.get
    args:
      url: https://www.example.com/MediumFunc
    next: end
- large:
    call: http.get
    args:
      url: https://www.example.com/LargeFunc
    next: end
```

Listing 3.4: Example Google Cloud Workflow. Source: https://cloud.google.com/workflows/docs/reference/syntax/conditions

3.3 Functional composition model

Functional composition models define a composition as a data stream and view the individual serverless functions as parameters, like higher-order functions.

3.3.1 OpenWhisk Composer

OpenWhisk Composer provides a library with combinator methods that can be used to create compositions. There are a variety of combinators that implement different programming constructs, like *let*, *map* or *sequence* for example. Compositions are built up from a single parent combinator. It is also possible to execute steps in parallel, but this requires access to a Redis instance that can store the intermediate data. Listing 3.5 shows an example of an OpenWhisk composition that returns success or failure depending on a password check. The example uses in-line JavaScript functions but if the actions already exist on OpenWhisk, their identifiers can be used instead.

```
const composer = require('openwhisk-composer')

module.exports = composer.if(
    composer.action('authenticate', {
        action: function ({ password }) {
            return { value: password === 'abc123' }
        }
    }),
    composer.action('success', {
        action: function () { return { message: 'success' } }
    }),
    composer.action('failure', {
        action: function () { return { message: 'failure' } }
    })
)
```

Listing 3.5: Example OpenWhisk composer workflow. Source: https://github.com/apache/openwhisk-composer

3.3.2 FaaS-flow SyncChain

FaaS-flow's SyncChain follows a programming model that is more similar to functional programming as a composition is modelled as a series of function applications on the composition input. However, the SyncChain only allows strictly sequential compositions so it is not nearly as expressive as its async counterpart. Listing 3.6 shows a SyncChain that applies two existing functions and modifies the output with an in-line function.

```
func Define(flow *faasflow.Workflow, context *faasflow.Context) (
    err error) {
    flow.SyncNode()
        .Apply("func1")
        .Apply("func2")
        .Modify(func(data []byte) ([]byte, error) {
            // do something
            return data, nil
        })
    return nil
}
```

Listing 3.6: Example FaaS-flow SyncChain. Source: https://github.com/s8sg/faas-flow

3.4 Orchestration and choreography for function composition

The aforementioned composition models only concern the APIs for serverless function composition. The back-end implementation can also be divided into categories that are well-known in web service composition [26]. These categories are orchestration and choreography.

In choreography, the individual components are responsible for calling the next step of the composition when they terminate. This allows the implementation of composition without the necessity for an external entity to control the composition. The disadvantage of choreography is that some form of middleware will have to be added to the components. Another disadvantage is that if one of the functions in a chain fails, the chain will stop and its state will be lost.

Orchestration works by introducing a separate component that is responsible for the co-ordination of the functions in a composition. The state of a composition invocation is generally stored in a persistent data store. The state of the invocation and the definition of the composition can then be used to determine the next step to execute. The orchestrator can also retry failed functions.
The mentioned serverless composition implementations all follow some form of the orchestration pattern. This makes sense because it does not require any changes to the serverless functions or the serverless platform themselves. There are, however, differences between the orchestrators on the different platforms. For example, Azure Durable Functions allows the definition of an orchestrator as a serverless function, while, for example, fission workflows uses its internal scheduler and controller components as orchestrators.

The implication of the distinction between choreography and orchestration is that a side-effect abstraction based on choreography can be made platform-agnostic, because it communicates from function to function in the in- and output bodies. A similar system based on orchestration requires changes to the orchestrator to make it aware of the abstraction. For this reason two prototypes of the state abstraction were developed. The first prototype uses a custom implementation of orchestrations, based on Azure Durable Functions, on the Fission platform. In the Durable Functions framework, the orchestrators are functions defined by the users, meaning the implementation of the orchestrator is very flexible. The custom implementation of Durable Functions is built on the monadic abstraction as well, and is explained in section 5.5. The second prototype is built on the UbiOps platform, which has a closed-source orchestrator running on their servers. This means the prototype needs to rely only on choreography to implement the abstraction.

Chapter 4

Monadic programming and serverless computing

This section explains the concept of monads in more detail and explores the applicability of monads in serverless computing.

The problem that serverless faces that prompted the research topic of this thesis is the inherent statelessness of serverless functions. Serverless functions have the ability to store state in memory or on disc but there is no guarantee any new function invocations will be executed in the same container. The state will be lost if the container is destroyed, and it is inaccessible for another function invocation that is executing on a different container.

Similarly, functional programming languages generally discourage the use of side-effects. Haskell even disallows side-effects entirely. The concept of monads was developed for Haskell to implement side-effects in a pure way. The question arises whether this concept can also be used in a serverless computing environment to transparently solve its statelessness. But first, the next section will explore what monads are exactly.

4.1 Monads

Monads originate from category theory, where they are endofunctors mapping a category to itself. They were first used to structure denotational semantics of programming languages by Eugieno Moggi [27]. They were later used to structure functional programs by Philip Wadler [28].

Monads are used by Haskell to make side-effects explicit by wrapping the return value of a monadic function in a monad wrapper. For example, the Maybe monad wraps a return value to allow a function to return *Nothing* or *Just* x, where x can be any value. The implementation of the maybe monad can be found in Listing 4.1.

The maybe monad removes the necessity of a null value, which is a type unsafe way to represent the same concept and has been called "the billion dollar mistake" by Tony Hoare, the inventor of ALGOL W in 1965 [29]. The Maybe monad is a simple data structure that wraps the return value of a function but the return value can also be wrapped in a function itself, which is how the state monad is implemented.

4.1.1 Monad properties

A monad definition consists of three components: a type constructor M, a type convertor (unit/return) and a combinator (bind). The type constructor M defines the monadic datatype M T with respect to another type T. The definition of the maybe monad is a class Maybe T with the instances Just T and Nothing.

A type converter converts a non-monadic value to a monadic value, it is called unit or return. Its type signature is $return : T \to MT$.

A combinator, the most common being called bind and denoted as (>>=), unwraps a monadic value and applies the next function on the unwrapped value. Its type signature is $(>>=) : Ma \to (a \to Mb) \to Mb$. Another common combinator is the *then* operator, which is denoted by $(>>)$. The *then* operator discards the return value of the first function and only applies the monadic component to the next function.

The implementation of the maybe monad in Haskell is shown in Listing 4.1[1].

```haskell
return :: a -> Maybe a
return x  = Just x

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
m >>= g = case m of
             Nothing -> Nothing
             Just x  -> g x
```

Listing 4.1: Maybe monad implementation [30]

The bind function is the essence of monadic function composition. Without it, every monadic value would need to be unwrapped using pattern matching and its combinator logic repeated, which would greatly reduce the readability and maintainability of the code. The maybe monad has a relatively straightforward combinator, but other monads can be significantly more complex.
The bind function is also the most interesting monadic concept for serverless because it defines rules for composition, and, for example, could be used to abstract away intermediate data storage from the end-user.

[1]:: denotes a type definition

4.1.2 Do notation

Haskell introduced the do-notation as syntactic sugar for a sequence of function calls that allows naming function results. This notation resembles imperative programming and variable assignment. The "variables" in do-notation assignments are however implemented as a bind with a lambda function, not as an actual variable. If the result of a function is not used by the next function, the variable assignment can be left out and the do-notation will be desugared to the *then* operator instead of bind. Listing 4.2 contains an example of a do-notation and its desugared form.

```
-- Do-notation example
do
    action0
    x1 <- action1
    x2 <- action2
    mk_action3 x1 x2

-- Do-notation example desugared
action0 >> action1 >>= (\ x1 -> action2 >>= (\ x2 -> mk_action3 x1
     x2 ))
```

Listing 4.2: Do notation example

4.2 Common monads

Haskell itself provides a library with monads for common side-effects, the most prominent of which is the IO monad. The IO monad abstracts input/output side-effects, like the interactive console or file-system interactions. The main function of any Haskell program is required to return an IO monad. However, the IO monad will not be explored because there are other monads that are more applicable to serverless computing.

Two examples of monads that may be useful in a serverless computing context are the **state** and **list** monads. The **list monad** represents a list and is commonly used for non-deterministic computations with functions returning all possible solutions in a list [31]. The unit function is simply defined by $unit\, x = [x]$. The type signature of the bind operator is $[a] \to (a \to [b]) \to [b]$ and is implemented as $xs >>= f = concat\,(map\, f\, xs)$. The bind operator maps over the input list with the next function and flattens the results by concatenating them into a single list.

The **state monad** allows functions to use a state that implicitly gets passed through a composition [32]. Its implementation is shown in Listing 4.3.

```haskell
newtype State s a = State { runState :: s -> (a, s) }

state :: (s -> (a, s)) -> State s a

return :: a -> State s a
return x = state ( \ s -> (x, s) )

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = state $ \ s0 ->
   let (x, s1) = runState p s0  -- Running the first processor on
       s0.
   in runState (k x) s1         -- Running the second processor on
        s1.
```

Listing 4.3: State monad implementation

The state monad wraps a function into another function that takes an initial state as input and produces a new state and a value.
The data type is defined in the first line of Listing 4.3. The constructor takes a function (s -> (a, s)) and stores it as *runState* using Haskell's record syntax[2].

The return function shows the simplest state wrapping. *return x* returns a state monad with a lambda that takes a state s, and returns the tuple (x, s) without modifying them.
The bind operator returns a state monad that takes an initial state s0 and first executes p with s0 using *runState p s0*. The function k is then applied to x and run with state s1.
The state monad provides the *get*, *put*, *evalState* and *execState*. *get* returns the content of the state, *put* sets the state, *evalState* executes a state monad and returns the value and *execState* executes a state monad and returns the final state.

4.3 Applicability of monads in serverless computing

Structuring side-effects using monads provides a number of benefits. The first benefit is the isolation of side-effect code. The implementations of standard monadic functions, like the *set* function of the state monad, are isolated in the implementation of the monad. Users do not need to concern themselves with implementation details when using standard monadic functions. Anyone creating monads themselves will also be able to use them to separate monadic logic from application logic cleanly.

Monads also make the presence of a side-effect explicit when looking at the source code of a function, because the monad is part of the return type. Anyone that sees a function returning a State monad can trivially conclude that the function uses some

[2]This means that given a state monad *p*, the function *runState* can be accessed with *runState p*. The function can then be executed with an initial state *s0* with *runState p s0*.

kind of state.

The next benefit is that monads make sure that compositions of monadic functions are well-defined, because composition is part of the monad. All monads can be composed in the same way using the *bind* operator, meaning users with an understanding of monads in general can use any monad easily.

Monads also provide a vehicle for a language or platform to provide standard implementations of common patterns, similar to how Haskell provides some monads in its standard library.

Each of the mentioned benefits would be valuable in a serverless environment. The isolation of side-effects can make testing serverless functions easier, because an isolated side-effect can be mocked more easily.
Making the side-effects of a function explicit to the platform creates the potential for the serverless platform to implement optimisations. For example, a function that uses an external state storage may be scheduled to run on a machine that is closer to the state store.
Finally, the focus on composability of monads is very valuable in a serverless environment as well. Function composition is very important for long-running analytics jobs due to the runtime restrictions of serverless functions. Monads are therefore a powerful tool to implement features like a state that is shared among a set of composed functions.

Chapter 5

Cloud monads

This section explains the developed abstraction model of cloud monads and the underlying implementations that support it. The section 6 will also explore a shared state implementation for serverless functions on the UbiOps platform as a use-case.

5.1 Monadic values

There are two types of monadic values considered in the abstraction. The first type is where the monad requires some interaction with an external service, like a centralised logging service, or an external data store. The monadic components then consist of configurations for interacting with a service. These configurations can consist of information like the address of the service, or an identifier that the function invocation should use during its interactions with the service. This type does not need to use the bind function explicitly, as the configuration is passed down to an orchestrator's child functions and does not change because the changes to the service state are stored in the external service itself.

The second type of monadic value only embellishes function output in some way, and does not interact with any external services. This implementation of monadic values is closer to how they function in Haskell, because the monadic value wraps the return value of the function. Monads that use this type of monadic value have to implement a *bind* function.

An example of the distinction between both types of monadic values is the state monad, which can use an external service to store the state, which would correspond to the first type, but can also share the state directly between function invocations using request and response bodies, corresponding to the second type.

There are a few ways that monadic values and configurations can be set in a function. The first is to simply add monadic input and output to the body of the request and the response of the function. Monadic functions look for the *monads* field in the input body. The *monads* field contains Key-Value pairs with the key being the identifier of the monad, and the value being an object containing the actual configuration of the monad.

Environment variables can also be used to set defaults for monadic components if the platform supports it. This is especially useful if the platform supports environment variables at different levels, for example at function, function version or composition level. Environment variables can provide a default monad configuration, with the possibility of overruling it using the request body.

The final option is not an existing features of serverless platforms, but a feature that is part of the developed cloud monads implementation on fission. Users can add monads to their functions by adding a JSON file to their function, containing a list of objects. Each object contains the identifier of the corresponding monad and any configuration options. Similar to environment variables, these configurations can be overruled using the request body.

5.2 Monadic function layer

Implicitly handling monadic components through the function body is achieved by adding a monad layer in the form of a function template. The monad layer acts as a proxy to the user code, allowing it to perform some initialisation, execute the user code, and handle termination. The user can interact with the monad layer through an API that is exposed through the context, which is available in most serverless offerings and contains some extra information about the function execution. A visualisation of the flow of the monad layer can be seen in Figure 5.1. Listing 5.1 shows how a monad can be accessed and used through the context object in a Fission function. The implementation of the monadic layer for Fission can be found in Listing B.1 in Appendix B.

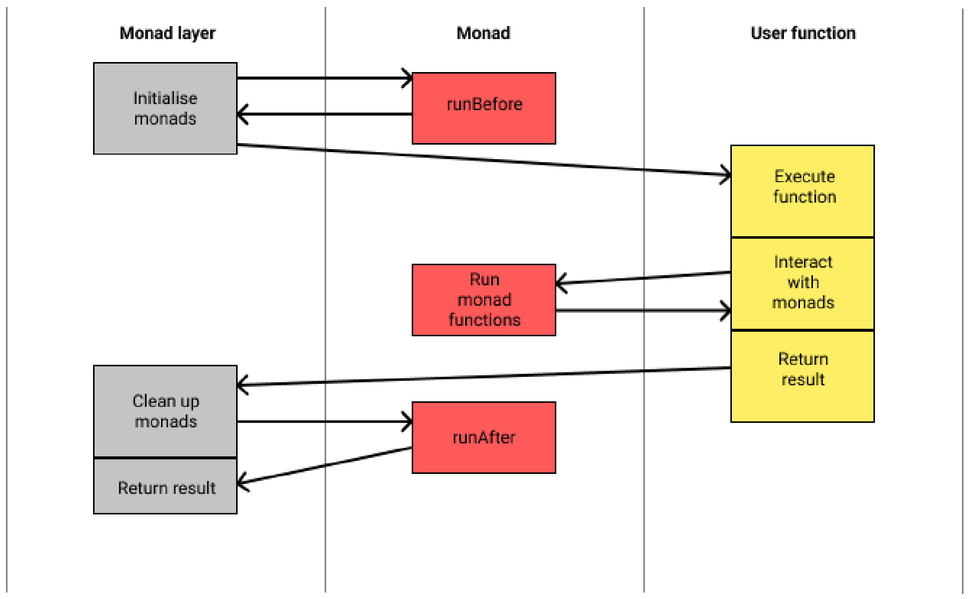


Figure 5.1: Flow diagram of a function using the monadic function template.

```javascript
module.exports = async function(body, context) {
    // Get logging API and function input
    const logging = context.monads.logging;
    const input = body.st;

    await logging.log(`Started work with ${input}`);

    // Sleep to emulate work being done
    await sleep(1000);

    await logging.log('Finishing up');

    return {
        st: `${input} +`
    };
}
```

Listing 5.1: Function using logging monad

The monad layer is added to the function through an extra build step. The build step is executed by the user before uploading a function to the platform but could also be a part of the function deployment pipeline of a platform. The build step adds the monad code for the monads in the monads JSON file to the function. The build step also takes the target platform as input parameter, with current implementations available for Fission and Azure Functions.

5.3 Monad implementation

The monad layer interacts with the monad using a number of functions. These functions should all be implemented for the monad to function. These functions are: *runBefore*, *runAfter*, *bind* and *createMonadBody*.

runBefore and *runAfter* are executed before and after the user function, which is similar to the concept of join points in aspect-oriented programming. They both receive the context as parameter, *runBefore* also receives the input body, so it can read the monadic values. *runBefore* adds the monad API object to the context object to give the user access to it. *runAfter* adds the monadic components to the result of the function.
createMonadBody is a function that allows user to explicitly initialise a monadic value component. This monadic value component can then be used as a parameter for a function call from an orchestrator. This function is similar to the monadic *return* function, because it can be used to wrap an arbitrary value into a monadic value. The custom composition framework also implements the monadic *return* function so it can be used as a starting point for composing functions using *bind*. The function is called *initBind* instead of *return* to avoid confusion with the imperative keyword return. *initBind* uses the *createMonadBody* function of the monad specified in its parameters to create the monadic value.

Some monads only work using the *bind* function. The list monad, for example, uses the *bind* function because it needs to launch multiple functions. This can only be implemented with a *bind* function that is executed on the orchestrator. The *bind* function is defined in the monad but has access to the composition framework so it can launch other functions.
The *bind* function can also be used indirectly by invoking the *do* function, which is the equivalent of the do-notation in Haskell. The *do* function takes a sequence of function identifiers and returns a function that can be invoked with the initial function input, its monadic body and the name of the monad in which context the bind will execute.

Listing 5.2 shows how *bind* and *do* can be used. Lines 7 to 10 show a sequence of binds. The sequence is started using the function *initBind* of the orchestration monad, which wraps the input in the first parameter (the number 1) inside the monad identified by the second parameter (the list monad). The final parameter contains the monad configuration, and is unnecessary here because the list monad does not require any configuration. Lines 16 to 23 show the usage of the *do* function. On line 16, the *do* function is called using an array of function identifiers. It returns a function that is called on line 23 with the initial input, any monad configuration and the identifier of the monad.

```
module.exports = async function(body, context) {
    // Retrieve the orchestration monad from the context object
    const o = context.monads.orchestration;

    // Calling the function 'list-test' multiple times using bind.
    // list-test takes a number as input and returns an array
        containing the input multiplied by 2, 3 and 5.
    let res = o.initBind({num: 1}, 'list', undefined);
    // res contains the monadic value [1]
    res = await res.bind('list-test');
    // res contains the monadic value[2, 3, 5]
    res = await res.bind('list-test');
    // res contains the values [4, 6, 10, 6, 9, 15, 10, 15, 25]

    // Calling list-test multiple times using a do function.
    // The do function returns the composition as a function that
        can be called to execute the sequence.
    const doFunc = o.do([
        'list-test',
        'list-test'
    ]);

    // The parameters of the do function are the input body, any
        monadic values that should be added and the monad in whose
         context the binds should be executed
    const doResult = await doFunc({num: 1}, undefined, 'list');
    // doResult contains the values [4, 6, 10, 6, 9, 15, 10, 15,
        25]

    return 'Finished the orchestrator!';
}
```

Listing 5.2: Example of an orchestrator function and the different ways to call other functions

5.4 Using monads in different composition frameworks

Cloud monads has been implemented on two platforms; Fission and UbiOps. The implementation on Fission uses a custom composition framework that is based on Azure Durable Functions. The composition framework is built with monadic composition in mind, more details about it can be found in section 5.5. The orchestrator pattern gives the most flexibility and control in terms of injecting monadic components into descendant function calls. The bind function, for example, is only feasible if it can be executed on the orchestrator.

There are a number of ways for the user to pass monads to children of the orchestrator. The first is to let the orchestrator do it implicitly. If a child function call does not receive monadic configurations, all configurations of the orchestrator will be shared with the child. For example, for state this means it will operate in the same shared state as the orchestrator, and all other children with default configurations. The monadic configuration can also be added to a function call explicitly through a separate monad parameter and the *exportMonad* function of the corresponding monad. An example of an orchestrator is shown in Listing 5.3. Line 7 uses *callActivityAsync* to call the function *logging-test* with the value of the second parameter as input. It does not provide a third parameter so the monadic context of the orchestrator is implicitly shared with the function. Line 15 shows a function call where the monad body that was defined in line 12 is used to call a function in a different context.

```
1   module.exports = async function(body, context) {
2       const logging = context.monads.logging;
3       const durable = context.monads.durable;
4
5       // Call function with logging monad, implicitly passing monad
            configuration of orchestrator
6       await logging.log('Calling function with call activity async
            and waiting for it.');
7       const result1 = await durable.waitFor(durable.
            callActivityAsync('logging-test', {st: 'function-input'}))
            ;
8       await logging.log('Function returned: ' + result1);
9
10      // Call same function with logging monad, explicitly passing a
             monadic configuration with identifier "invocation-2"
11      await logging.log('Calling function with call activity async
            and waiting for it.');
12      const logMonad = {
13          logging: logging.createMonadBody('invocation-2', logging.
                host)
14      };
15      const result2 = await durable.waitFor(durable.
            callActivityAsync('logging-test', {}, logMonad));
16      await logging.log('Other function returned: ' + result2);
17
18      await logging.log(`Finished the orchestrator!`);
19
20      return 'Finished the orchestrator!';
21  }
```

Listing 5.3: Example of an orchestrator function

If a monadic function is called without monadic configuration in the body of the request, a default configuration is used. The orchestrator function also generates a default configuration for any monad without configuration, which it will send to any child invocations it calls. Any monadic function will always return the monadic configuration it operated in.

The implementation of cloud monads on the platform of UbiOps is different because of being restricted to their composition framework, pipelines. The only mechanisms for setting monadic configuration are environment variables and the request body. If there is a request body, the monadic layer uses that, otherwise it uses the environment variables if they exist. If neither exist, the function fails. Pipelines model a sequence of functions. The only way to pass monadic configurations from one function to the next is by adding them to the request body. This unfortunately means all functions in the pipeline will have to pass the configuration to make sure it's not lost, even if the function itself does not use it.

5.5 Custom composition framework

A custom composition framework was developed based on Azure Durable Functions that allows users to create orchestrator functions that can call other functions. Orchestrator functions themselves are built on the developed monadic abstraction as well. It uses a monadic abstraction for state and MQTT[1]. It also requires a the MQTT Trigger service. The MQTT trigger subscribes to the MQTT broker, executes the functions that are published there, sets the results of the function invocations in the state abstraction and restarts the orchestrator.

The monadic abstraction for state is an API for a simple, external key-value store that is used to store the progress and results of child functions of an orchestrator.
The monadic abstraction for MQTT sets up a connection to an MQTT broker based on the monadic configuration. The user can then register listeners and send messages to other processes connected to the same MQTT broker.
The MQTT trigger mechanism allows functions to be invoked by publishing a message with its parameters in a specified MQTT topic, instead of using an HTTP endpoint. This enables asynchronous function execution as no connection needs to stay open while the function runs. The result of a function is then set in the state of the orchestrator invocation and the orchestrator is resumed. Using MQTT to invoke functions also makes the orchestrator itself platform-agnostic. The MQTT Trigger is the only component of the composition framework that is dependent on the platform.

Similar to Azure Durable Functions, orchestrator functions do not wait for its child functions to finish executing. When a child function is called, the orchestrator exits, returning its invocation id. When a child function is finished, the orchestrator will be restarted using the invocation id to continue the function. Figure 5.2 shows a visualisation of the life cycle of an orchestrator function.

[1]MQTT is a lightweight, publish/subscribe messaging protocol

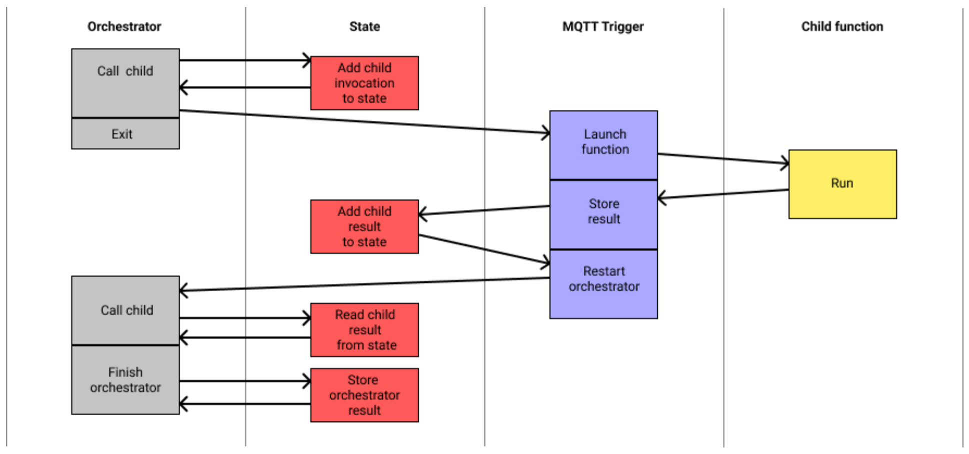


Figure 5.2: Flow diagram of an orchestrator calling a child function.

When the orchestrator starts a child function, it saves the fact that a child invocation has been called in the state and publishes a message on the message queue trigger containing the parameters of the function call. The composition framework implicitly adds the monadic configurations of the orchestrator to the child invocations. Some information about the orchestrator is also added to the message so it can be restarted by the message queue trigger when the child invocation terminates.

When the orchestrator needs the result of a function invocation, it checks its state to find the status of the invocation and the result if it is finished. It calls the function and terminates if it has not already been started, it terminates if the function has been started but has not been finished, and it continues if the function has finished.

The result of the orchestrator is stored in the state when the orchestrator successfully terminates. The state can then be queried using an HTTP endpoint using the invocation id of the first orchestrator invocation to get the result. The orchestrator can also be invoked using the original invocation ID to get the result.

The orchestrator should not contain side-effects itself if they should only be executed once. The replays of the orchestrator will execute them again otherwise. Orchestrators also have to be deterministic or the replays of the orchestrator will be unpredictable, which is a restriction inherited from Azure Durable Functions.

Chapter 6

Cloud monads for function state on UbiOps

One of the most interesting instances of side-effects for serverless computing is a state that is shared across invocations. There are a plethora of use-cases that become significantly easier to implement if there is an abstraction of state in place. It also allows developers to focus on their business logic, instead of having to worry about issues like scalability and consistency of their state storage. A shared state side-effect was implemented on the platform of UbiOps.

6.1 Design goals

The API for the state abstraction should be as simple as possible while still striking a good balance between usability and expressiveness. Ideally the data in the state can be interacted with in a way that is as similar as possible to interactions with local data.

Consistency is a difficult consideration in a shared state for a serverless environment due to the high degree of elasticity. Ideally the user does not have to worry about consistency when using the abstraction, or at the very least, does not have to implement any measures to ensure it themselves. The use of atomic, incremental operations can make sure that the state stays mostly consistent without complex data versioning and data version dependent operations.

6.2 Data types and operations

The state abstraction is based on a number of data types and operations. The top-level storage is a bucket. A bucket has an id that can be used to access it. A bucket is a map that can hold key-value pairs. There are a set of common operations and some specific operations for each datatype. Trying to apply an operation a non-

Data types	Operations			
All types	Get	Set	Exists	Delete
Boolean	Invert			
String	Prepend	Append	Get length	
Number	Increment	Multiply		
Map	Get keys			
List	Get size	Append	Trim	Is empty

Table 6.1: Categorization of composition frameworks

existing key or to the wrong type will result in an error. An overview of the data types and operations that were implemented can be found in Table 6.1.

Every field stored also contains a field with metadata. The metadata has been added so the data types can be extended in later development. An example of a possible extension would be to specify a maximum length in the list which would ensure a list stays the specified size when appending.

6.3 API

The way a user can interact with the state is by executing operations. An operation is defined at least by an operation identifier and the key of the data to operate on. An example of such a function is the list length operation, which returns the length of a list specified by the data key.

Some operations require a value as well, and may alter the contents of the state. For example, increment number takes an optional value to define by how much a number should be incremented. The value field can also specify another value in the state to use as value parameter.

There are also some operations that allow additional behavior by providing optional options. The function "exists", for example, has an optional parameter "otherwise", which allows the user to initialise the field if it does not exist.
An example of a deployment using the shared state can be found in listing B.2 in appendix B. This deployment uses the shared state to return a moving average of the last 10 inputs that the deployment has received.

6.4 Implementation

6.4.1 Back-end architecture

The architecture of the shared state back-end is separated into two layers; the API layer and the storage layer. This separation allows the API and the storage service to scale independently from each other. It also enables the storage service to be interchangeable depending on the user's requirements.

The API layer has two endpoints for registering and unregistering buckets, one endpoint for executing an operation and an endpoint for executing a transaction, which consists of a list of operations. The API layer also handles user authentication through API keys that need to be added to the requests by the client. The API layer connects to the storage layer through an interface that defines reading and writing operations.

The storage layer consists of a storage medium, like Redis or SQL, and an adapter that implements the actual storage interactions the API layer needs. This separation allows the storage medium to be interchangeable, depending on the requirements of the user.

The developed prototype combines the API and storage layers into one service for simplicity but in a complete implementation these should be separated. The developed prototype state service is a Node.js application built on the express framework. It stores the users' state in memory. An in-memory storage has obvious limitations with regards to scalability of the amount of data that can be stored, but works well for applications that require low latency and only have a constant, small memory footprint.

6.4.2 Client library

The client can access the shared state through a small client library that interacts with the back-end via HTTP requests. Each operation has a corresponding function that executes it. Each operation also has a function that returns the operation as a python dictionary. An array of these dictionaries can be used to execute a batched operation, reducing the amount of round-trips to the back-end. A PyDoc document of the python client library can be found in appendix A.

6.4.3 Developing and debugging

The state service can also be made accessible from other sources outside of functions. This way the state of a function can be viewed live and issues can be debugged while the function is deployed. It is also possible to run a function locally during development, while using a deployed state service that is equivalent to one running in production using a local Docker container.

Chapter 7

Evaluation

The monadic abstraction and the shared state prototype have been evaluated on three properties: performance, expressiveness and usability. The three properties are evaluated in their respective section.

The performance section explores the execution times of the build step of the monadic abstraction and the time the monadic abstraction requires to set everything up at the start of the function. The state prototype is also evaluated by measuring the latency between state client running on UbiOps and the back-end deployed on Google Compute Engine. The scalabality with many concurrent functions is also evaluated.

The expressiveness section looks at the applicability of the monadic abstraction to different side-effects. It also explores the applicability in different programming languages and FaaS offerings.

Finally, the usability section evaluates the usability and contains the results of an internal evaluation of the shared state at UbiOps.

7.1 Performance

The following section evaluates the performance of cloud monads and the state prototype.

7.1.1 Monadic function build time

The monadic abstraction implementation of Node.js uses a build step to include the side-effect abstraction in the user's function. The execution time of a build step only impacts the time to deploy so requirements are not as tight as if it would influence run time. The runtime of the build step was evaluated on a windows machine with an i7 8700k CPU and an SSD. The measured build time is 8.59 $\pm$ 2.06 ms[1] after 100 runs. The build step uses local files as source for the monads but if some kind of remote repository is used for retrieving them, the runtime will increase.

7.1.2 Monad set-up time

Once a function has been deployed, any invocation will start with setting up the monadic components of the function. This does factor into the execution time of the function so the overhead should be moderate. The runtime of the build step was evaluated on Fission running on a local Kubernetes cluster on a windows machine executing a function with a single monad, and an orchestrator function using five monads. The measured set-up time of the single monad function is 0.11 $\pm$ 0.02 ms. The measured set-up time of the multiple monad function is 0.36 $\pm$ 0.09 ms. Both were executed 100 times.
The relative impact of this step on the total runtime of a function depends on the amount of work executed in the function but it should not cause significant delays. If monads use their set-up function to establish an active connection to an external service the set-up step may also take longer depending on the quality of the connection to the service.

7.1.3 State prototype response time

The response time of the state prototype was evaluated using a UbiOps deployment[2] that implements a distributed moving average. The deployment uses the state to keep a buffer of recent inputs. When the function is called with a new input, the input is appended to the state, the buffer is trimmed to a constant length and the deployment gets the current buffer to calculate the average. Essentially this executes three state operations: *append*, *trim* and *get*. These operations are then batched into a single call to the state service to reduce round-trip times. This is possible because all three operations will always need to be executed, i.e. there is no conditional execution that is implemented by the deployment.

The response time of state requests are measured in the deployment and the execution time of the operations is measured in the back-end. The deployment that was used can be found in listing B.2 in appendix B. The runtime of the batched operation on the back-end is 1.10 $\pm$ 0.36 ms while the response time of the batched operation measured by the UbiOps deployment is 45.45 $\pm$ 3.71 ms, after 100 runs. These results show that the response time of a call to the state prototype is dominated by the network request. Without batching the state interactions, the negative impact of the network overhead on the runtime of the deployment would be even

[1]Execution times are denoted as (mean $\pm$ standard deviation)
[2]"Deployment" is the functionally equivalent UbiOps naming of a serverless function

greater.

7.1.4 State prototype scalability

The scalability with regards to storage size is poor in the prototype because it stores user state in memory. The amount of requests it can handle per second was evaluated using Locust, an open-source load testing framework [33]. The request used were the batched operations of the moving average implementation mentioned in the previous paragraphs. The state service was running on an 'e2-small' VM instance on Google Compute Engine [34]. The response times for an increasing amount of requests per second can be seen in Figure 7.1. The 95% percentile value for response time spikes significantly around 2200 requests per second, implying that is the amount of load that the service can comfortably handle. The requests per second also drops significantly but the response time does not recover.
The state service itself is very lightweight, as it can handle around 2000 requests per second without a significant increase in response time on a relatively weak CPU. The increase in the 95% percentile value of the response times at 2200 requests per second suggests that the CPU usage nears 100% around that amount of load.

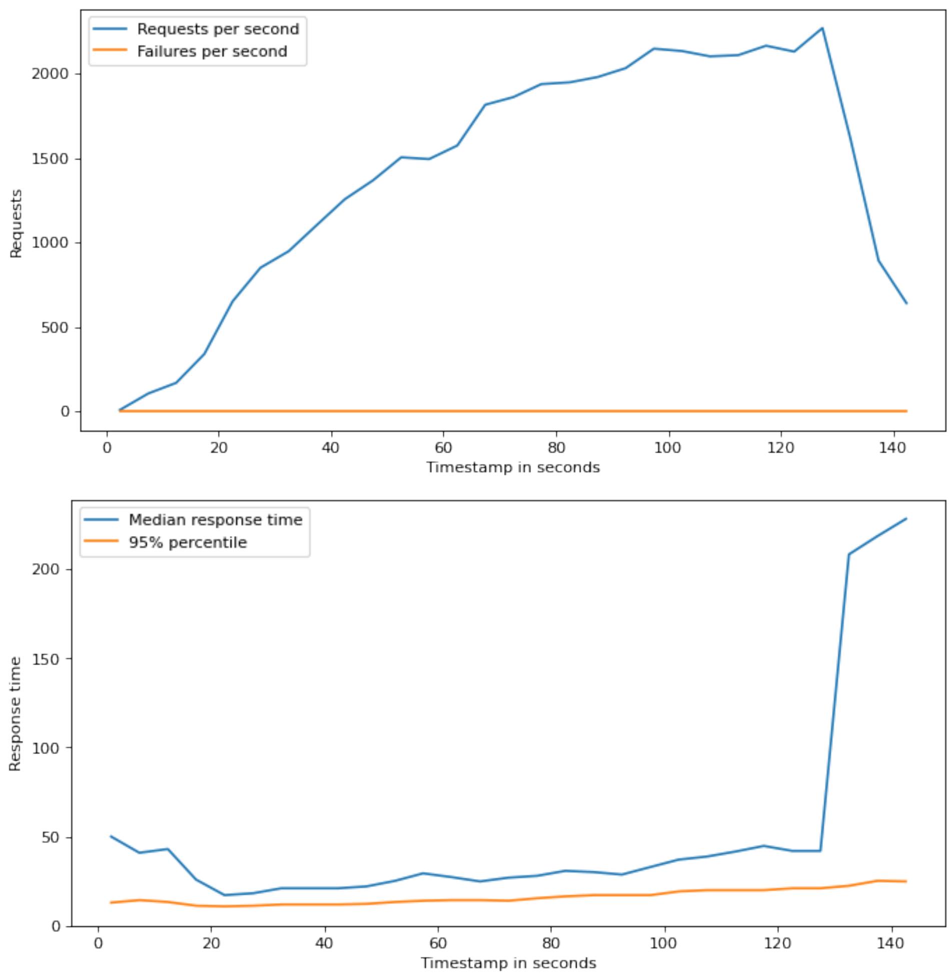


Figure 7.1: The amount of requests per second, and the median and 95% percentile response times in ms.

7.2 Expressiveness

The following section evaluates the expressiveness of cloud monads.

7.2.1 Haskell monads in the cloud

The possibility to implement existing Haskell monads in serverless is the first measure of expressiveness used to evaluate the developed side-effect abstraction. The Haskell monads that are explored are Maybe, List, State, Reader and Writer.

The maybe monad can be implemented by adding a monadic component that contains the type Nothing or Just. The implementation of bind is straightforward, calling the next function if the the monadic component of the previous value is Just, otherwise returning Nothing. The users will have to explicitly specify it if they wish the function to return Nothing. A potential improvement would be to add an *onException* to the abstraction that can be used to automatically return Nothing if the function fails.

The list monad is an example of a function that requires bind. The list monad is generally used to model non-deterministic computations. The bind function runs a function for all elements of the list. In the serverless environment it needs to have access to the composition framework so it can launch functions. The list monad has been implemented on Fission, but it can not be implemented within the UbiOps pipeline system because it only supports sequences. It would be possible to implement the list monad on UbiOps by using the HTTP endpoints for deployments directly but this would incur double-billing, because the original deployment will have to stay alive to wait for the other deployments to finish.

The state, reader and writer can be considered very similar to each other with regards to their implementations because reader and writer are conceptually subsets of state. The reader monad implements a state where reading is the only available operation. The writer monad implements one where writing is the only available operation. These monads can be used without using the *bind* function if an external service is used, because the monadic components only contain a reference to the service and are not changed by child functions in that case. Without external service *bind* is required, because the monadic components returned by a function will contain the state.

7.2.2 Applicability to different programming languages and FaaS offerings

One of the goals of the side-effect abstraction was to ensure it can be applied in different languages and on different platforms. It is infeasible to create implementations for all languages and platforms so a a small subset of diverse options was chosen. Implementations of the monadic layer were created for Fission in JavaScript and Java, and for UbiOps in Python. While JavaScript and Python are quite similar in the sense that they are scripting languages, the implementation in Java shows that it is also possible to implement the monadic layer in a compiled language with a strong type system and static type checking.

Fission and UbiOps were chosen as examples of opposites in terms of flexibility of the composition framework, with UbiOps' pipelines being closed source and supporting only sequences and Fission using a custom composition framework. The implementation of Fission is more complete and flexible in terms of usage patterns.

The other platforms should have comparable problems to the UbiOps implementation with respect to the monads requiring a *bind* function in a composition. However, because the monadic components are added to the bodies of the response and request, monads using an external service will still be able to function in any composition framework.

This problem only holds when using the composition solution of the platform. It is also possible to use a port of the custom composition framework developed in this thesis to define compositions instead, which would enable the patterns provided by *bind* again.

7.2.3 State prototype use-cases

The expressiveness of the state prototype was evaluated by doing a small study at UbiOps about use-cases that their customers may have for a shared state implementation. From the results it was possible to group the use-cases into four classes.

The first class of use-cases is sliding window models. Sliding window models use a recent inputs to draw a conclusion about a stream of values. A simple example is a moving average. The state that is stored in this use-case is a buffer of recent values. Buffers are a good fit for the developed prototype as they are constant in size. Another application that was mentioned during the study was the smoothing of sensor input using recent values.

Another proposed use-case was the implementation of a feature store in the shared state. Features can be stored in a bucket that can be shared by all functions that interact with them. These functions include those that generate the feature values and those that read them. The state service is currently limited by VM memory because it stores everything in-memory. This means it will be infeasible to store a complete history of data and feature values. However, if the feature calculation can be modelled as a stream algorithm, it is possible to keep a recent history that is discarded when the feature values are updated.

The state service would be suitable for ephemeral storage, although it does depend on the form of the data. Ephemeral data can be deleted explicitly after reading. Another possibility would be to add a Time-To-Live (TTL) or data persistence configuration feature to the state. The state also contains a combined read-and-delete operation.

The shared state can be used to tweak machine learning model parameters on-the-go, if the function retrieves them from the state at the start of execution. It would also be possible to update the model without having to redeploy any functions by storing the model in the state and retrieving it in the function. The data types supported by the state and the possibility to run operations on nested values may also make it possible to train incremental models with new values and update the model directly in the state using incremental operations on its weights.

The shared state can also be used to model some real-world entity.Two practical use-cases were encountered. The first use-case was an application that monitored parking garages and could recommend users looking for a spot to one. The state of parking garages would be kept in the shared state service. This system only performs recommendations so consistency requirements are less compared to a parking reservation system. The second use-case concerned an application that monitored patients in elderly homes through sensors in their bed. The basic implementation of both systems would need two functions that perform different roles. One function would update the state based on the sensor data. The other function would use the current state to make a recommendation.

The shared state can also be used to do monitoring of data. A data processing pipeline where anomalies are detected based on recent data and warnings or issues are stored in the state. Another function periodically checks the state for issues and takes action if necessary.

UbiOps identifies itself on the market as a solution for the deployment of machine learning models. The use-cases found are therefore mostly related to machine learning applications. However, a wider investigation with a more varied customer-base might provide more possible use-cases.

7.3 Usability

7.3.1 Monadic functions in isolation

The power of monads is mostly expressed in compositions. However, the monads that do not rely on the *bind* function to work also have value when functions are called individually. An example of such a monad is the state monad. Users will have to take care that they send the correct monadic body in a function request, and that they are aware what the default values are set to. Functions may cause unintended side-effects if a mistake is made, but the function returns the monadic values, which provides the possibility for a check and gives an indication about any incorrect state updates.

7.3.2 Composition

In a composition, the pass-by-reference style of monads does not require much input from the user because it handles the monad passing implicitly in the orchestrator. The default behaviour can be overridden to facilitate different usage patterns. The behaviour of the monads is summarised simply as a context that an

orchestrator shares to its child invocations.
The monads that require *bind* to function require more familiarity with the concept of monads and how they function in a composition. Additionally, it is also greatly dependent on the individual monad what the *bind* function does.
Similarly to Haskell, the do-notation is a more readable way to define a composition. The serverless do-notation implementation currently only supports a sequence of monadic functions that implicitly maps the output of each function to the input of the next function, but a more complete implementation would also support the assigning of variable names.

7.3.3 User-created monads

Monads in Haskell have a well-defined set of operators and functions that monads have to implement. This definition enables users to create and use their own monads. Cloud Monads were designed with a similar principle in mind.
Users are able to create their own monads for side-effects they require. These can be made very specific to the use-case for internal use in different functions, or they can be kept general so they may be shared and used by other users. Creating pass-by-reference monads is fairly straightforward, but monads that use *bind* also require access to a composition mechanism. This is doable if a particular composition implementation is assumed to be used, but otherwise becomes very difficult. It may be possible if a standard composition API is defined but it may not be possible for different composition implementations to use the same API.

7.3.4 State prototype user study

The usability of the shared state prototype was also evaluated internally at UbiOps with three of their employees. Two employees were back-end developers, one senior and one junior developer. The third employee worked in sales but also has programming experience, mainly in data science. They were tasked with using the shared state to implement two functions and a pipeline composing them. The first function calculates the moving average over recent inputs. The second function keeps track of the maximum value for the moving average and increments a counter whenever it is changed. The pipeline composes the two functions in a sequence. The environment variables of the functions were used to set the correct storage bucket for the functions to use.

The functionality of the first function was chosen to evaluate whether users would use incremental list operations to track recent inputs in a consistency-preserving way. Updating the list of recent inputs locally and setting it to the state directly would cause a race-condition, where two simultaneous updates would result in one of the updates being lost. The necessary operations can also be batched to optimise the amount of round-trips. The second function was added to evaluate whether it was clear how the monads were passed in a pipeline.

The importance of using incremental operations to retain consistency was not immediately clear. Instead, the participants' first instincts were to set values directly. It was also not inherently clear that data needed to be instantiated, or that it was possible to retrieve or initialise with a single operation.
The fact that the data types were close to programming language primitives was considered pleasant to work with.
It was not immediately clear how the state configuration in the input of the function related to the state configuration in the environment variables. This caused some confusion about how they should be used. The distinction is that the state configuration in the function input can be used to dynamically change the configuration of the monad for each request, while the environment variables can be used as a static configuration. The configuration in the function input is useful for compositions.
Some practical considerations were also raised about the in-memory state, especially how it's not suitable to any use-case that requires persistent storage. However, it was already built with the idea of extending it with multiple storage options for different use-cases so this is to be expected.

The participants of the user study were generally positive and saw the value of providing stateful functions as a feature on UbiOps. The intended customers of UbiOps are data scientists with limited software engineering and DevOps skills, so providing a state service that is tailored to the platform enables them to implement a wider range of applications.
Users are encouraged to use incremental state updates, but the API still allows "unsafe" operations like setting the state directly. One of the results of the user study is that it may be better to remove unsafe operations entirely, because users will still use them because they are conceptually more straightforward to work with.
Sending operations in batches can be implemented in a more user-friendly way, as suggested by one of the participants. They suggested to allow the user to create a batch object, which has the same methods as the state service, but which defers execution of the operations until a *commit* function is called.

Chapter 8

Discussion

There is a conflict between usability and faithfulness to the implementation of monads in Haskell. The pass-by-reference style monadic values are conceptually simple when used in an orchestrator but they do not resemble monads in Haskell very well. The pass-by-value monadic values resemble monads more closely, but they do not scale well due to having to send the monad in the request and response bodies.

Monads in Haskell often represent a type of deferred execution. Monadic functions do not return their results but instead return an 'action'. The state and IO monad for example represent actions that can be executed by using the *runState* and *runIO* functions. However, this does not fit the serverless platform well because a function can not return another function, and functions can not be dynamically created or changed.

One of the advantages of monads is the possibility for users to implement their own monads. The monad interface will then make sure that the monad is compatible with the monadic encapsulation. Creating custom monads is possible in the framework developed in this thesis but monads using the bind function are more difficult to implement due to the dependence on the composition monad. A complete solution would also contain some kind of repository for monads that users can use to share their own monads and find those created by others.

The basis for the composition framework was kept separate as a monad from the monadic encapsulation. The state monad that the composition monad relies on is also separated. This enables the potential for a better composition monad to be implemented on the same monadic encapsulation. However, making the composition part of the monadic encapsulation, or state part of the composition, may make some aspects of composition easier. For example, it may be interesting to explore a solution based on choreography, if the monadic encapsulation is added to all functions.

Monad transformers can be used in Haskell to compose monads. For example, a list of maybe values. The implementation of *bind* presented in this thesis does not support binding over multiple monads. Calling monadic functions directly does share all the monads of the orchestrator so the child function is able to inherit multiple monadic configurations. Monad transformers were left out of the implementation due to time constraints, but would be a great future improvement.

The presented composition framework is very basic and does not contain any response to functions failing or timing out. In those cases, the composition will simply stop. The state of the composition is still available in the state monad so a component could be introduced that watches the states of orchestrators and takes appropriate action.
It is currently also impossible to call other orchestrator functions from an orchestrator.

8.1 Related work

This thesis does not aim to solve the problem of stateful serverless, as much as it presents a model for interacting with similar features that can be considered "side-effects". However, stateful serverless is one of the most interesting applications and other stateful serverless solutions could be used as storage back-end for cloud monads.

Microsoft has published a paper on Azure Durable Functions and its semantics [35]. Durable Functions contains an implementation of state in the form of Stateful Entities. The entities and their operations can be defined by the user. Operations are viewed as events and ordered to determine the current state with eventual consistency. Orchestrator functions are also a form of stateful serverless, as the state of an orchestration is stored externally as well.

Crucial implements a distributed shared memory layer for serverless functions that can be used for state management and synchronisation [36]. Their Distributed Shared Objects (DSO) datastore allows developers to synchronise local variables with the distributed storage layer by using the "shared" annotation. The main programming abstraction of Crucial is a "cloud thread" which functions identically as a regular thread from the developer's perspective but executes on FaaS.

Boki is a serverless runtime that implements shared logs for serverless functions [37]. Boki provides ordering, consistency and fault tolerance, by having a "metalog" for all log records that stores the internal state transitions of the logs. The authors have used Boki to implement fault-tolerant workflows, durable object storage and message queues. The architecture of Boki is based on the FaaS system

Nightcore [38] and extends it with a logbook engine, storage nodes for log data, sequencer nodes for updating metalogs and a control plane to track system health.

CloudBurst implements a distributed data layer in the form of a key-value store that serverless functions can use for storage [39]. Cloudburst uses local data caches on function executors to have data locality where possible.

Triggerflow is an architecture for trigger-based orchestration of serverless workflows [40]. It follows an **Event Condition Action** architecture that is extendable on all levels. Workflows and DAGs can be mapped to triggers. For example, a DAG can be converted to triggers by creating a task termination event and trigger for each edge between tasks.

The "Serverless" framework [41] is an effort to make serverless as easy to use as possible. It abstracts away a lot of the boilerplate for serverless functions and also supports plugins that extend the functionality of the framework and of the functions created using the framework. These plugins are similar to cloud monads in the sense that they provide additional functionality to serverless functions in ready-to-use packages.

Chapter 9

Conclusion

A monadic abstraction for side-effect was developed and evaluated on Fission. An implementation of a shared state side-effect was also implemented for UbiOps. The implementations were used to answer the following research questions.

RQ1: Would monads offer an effective model of abstraction to simplify the implementation or use of side-effects in a serverless computing environment?
Concepts like state, logging or messaging can be implemented as monads. The user does not need to know the underlying implementation and can simply use the API provided by the monad through the function context. This assumes the definition of side-effects in serverless means interactions with external services. A definition that is closer to monads in Haskell requires the monadic programming style using bind and do-notations. However, this programming style is not very familiar to people that are accustomed to object-oriented or imperative programming styles.

RQ2:Would monads be able to provide a standardised solution for serverless function composition?
Function composition can be modelled using orchestrator functions that are built on the monadic abstraction themselves. This solution does require an external component that can invoke other serverless functions lives outside of the abstraction. Composition does not require invoked functions to know about the composition.
The abstraction is not bound to a single platform or programming language. It can also be used as a means to make a function platform-independent. It does require a mechanism for asynchronous function execution.

RQ3: Can monads be used in serverless to model and control side-effects?
Abstracting the configurations of external service interactions away from the user is an opportunity for serverless platforms to become more easy to use. It also enables the implementation of concepts like remote state in a way that is transparent to the user with regards to it being remote. It also promotes re-use of functions due to the configurability of the monad. Developing is also easier, as a function can

easily be configured to run in a certain environment.

RQ4: What infrastructure would be required to enable cloud providers to provide users with side-effects on their platform?
The state side-effect on UbiOps was developed as a means to explore this question. The developed shared state prototype is effective at functioning as a low-latency state store for states with consistent size, like buffers or the state of some real-world entity. It does not function well for infinitely growing states, like a persistent history because the service will run out of memory quickly. However, this can be solved by using a different storage solution for the service.
In the case of platform-managed state storage, some infrastructure would be required between the client code in the serverless function and storage solution. This layer would have to take care of authentication and billing. If the side-effects connect to user-operated infrastructure cloud providers would not have any extra infrastructure to operate except potentially a component in the scheduler that takes care of optimisations based on monads.

9.1 Future research

A monadic abstraction of side-effects makes the interactions with external services of a function explicit. A possible topic for further research would be to investigate if and how this information can be used to optimise the performance of the function. Potential parameters to optimise are the connection quality between a possible function placement and the external service, or warming certain functions for an orchestrator function.

Another topic for future research could be to create a domain-specific language for function composition that is based on monadic composition. This could preserve the clean declarative nature of the monadic programming style. The orchestrators presented in this thesis contain a lot of *await* and are bound to the object-oriented nature of JavaScript. A domain-specific language would allow most of this boilerplate code to be removed.

Serverless function composition frameworks generally do not employ type checking. However, this would be a valuable addition when developing compositions. A future topic of research could be how to implement a static type checker for serverless functions. Functions would need to specify their input and return types, and the static type checker could access that information to find type errors.

Functional programming languages generally work with immutable data. A possible topic of future work could be to make cloud monads immutable to reap the same benefits that functional programming languages enjoy due to this restriction. It would be interesting to evaluate the potential benefits against the overhead of data duplication in terms of storage size and state update latency.

Bibliography

[1] Youngbin Kim and Jimmy Lin. Serverless data analytics with flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 451–455, 2018.

[2] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 789–794, Boston, MA, July 2018. USENIX Association.

[3] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra, 2018.

[4] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[5] Aws step functions homepage. `https://aws.amazon.com/step-functions`. Accessed: 08-10-2021.

[6] Fission/fission-workflows: Workflows for fission: Fast, reliable and lightweight function composition for serverless functions. `https://github.com/fission/fission-workflows`. Accessed: 08-10-2021.

[7] Azure durable functions overview. `https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview`. Accessed: 08-10-2021.

[8] Configuring lambda function options - aws lambda. `https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html`. Accessed: 25-4-2021.

[9] Amazon ec2 instance types - amazon web services. `https://aws.amazon.com/ec2/instance-types/`. Accessed: 25-4-2021.

[10] Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. *2012 IEEE Fifth International Conference on Cloud Computing*, 2012.

[11] Phani Kishore Gadepalli, Gregor Peach, Ludmila Cherkasova, Rob Aitken, and Gabriel Parmer. Challenges and opportunities for efficient serverless computing at the edge. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 261–2615, 2019.

[12] Aws lambda resource limits. `https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html`. Accessed: 08-10-2021.

[13] Azure functions resource limits. `https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale`. Accessed: 08-10-2021.

[14] Openwhisk resource limits. `https://github.com/ibm-cloud-docs/openwhisk/blob/master/limits.md`. Accessed: 08-10-2021.

[15] Garrett McGrath, Jared Short, Stephen Ennis, Brenden Judson, and Paul Brenner. Cloud event programming paradigms: Applications and analysis. *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, 2016.

[16] Fission function triggers. `https://fission.io/docs/usage/triggers/`. Accessed: 08-10-2021.

[17] Openwhisk triggers. `https://openwhisk.apache.org/documentation.html#programming-model-triggers`. Accessed: 08-10-2021.

[18] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association.

[19] Per Persson and Ola Angelsmark. Kappa: serverless iot deployment. In *Proceedings of the 2nd International Workshop on Serverless Computing*, pages 16–21, 2017.

[20] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[21] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 12–1–12–55, New York, NY, USA, 2007. Association for Computing Machinery.

[22] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1992.

[23] Ankush Verma, Ashik Hussain Mansuri, and Neelesh Jain. Big data management processing with hadoop mapreduce and spark technology: A comparison. In *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, pages 1–4, 2016.

[24] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. Dague: A generic distributed dag engine for high performance computing. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1151–1158, 2011.

[25] Lyes Bouali, Karima Oukfif, Samia Bouzefrane, and Fatima Oulebsir-Boumghar. A hybrid algorithm for dag application scheduling on computational grids. In *International Conference on Mobile, Secure and Programmable Networking*, pages 63–77. Springer, 2015.

[26] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. Using application knowledge to reduce cold starts in faas services. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 134–143, New York, NY, USA, 2020. Association for Computing Machinery.

[27] E. Moggi. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, 1989.

[28] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, page 61–78, New York, NY, USA, 1990. Association for Computing Machinery.

[29] Null references: The billion dollar mistake. `https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/`. Accessed: 23-11-2021.

[30] Haskell - understanding monads - maybe. `https://en.wikibooks.org/wiki/Haskell/Understanding\_monads/Maybe`. Accessed: 23-11-2021.

[31] Haskell - understanding monads - list. `https://en.wikibooks.org/wiki/Haskell/Understanding_monads/List`. Accessed: 24-11-2021.

[32] Haskell - understanding monads - state. `https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State`. Accessed: 24-11-2021.

[33] Locust - a modern load testing framework. `https://locust.io/`. Accessed: 25-1-2021.

[34] Machine families | compute engine documentation | google cloud. `https://cloud.google.com/compute/docs/machine-types`. Accessed: 25-1-2021.

[35] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Durable functions: Semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[36] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. Stateful serverless computing with <span class="smallcaps smallercapital">crucial</span>. *ACM Trans. Softw. Eng. Methodol.*, 31(3), mar 2022.

[37] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[40] Pedro García López, Aitor Arjona, Josep Sampé, Aleksander Slominski, and Lionel Villard. Triggerflow: Trigger-based orchestration of serverless workflows. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, DEBS '20, page 3–14, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Serverless: Develop monitor apps on aws lambda. `serverless.com`. Accessed: 12-4-2021.

Appendix A

State prototype documentation

UbiState

index

Modules

json os requests uuid

Classes

builtins.object
    UbiState

class **UbiState**(builtins.object)

```
UbiState(data)

This class contains the API for the state service.
The API can be used by calling this class' functions.
```

Methods defined here:

__init__(self, data)
```
Initialise the API object using the deployment input data.
The bucketId and host are automatically extracted from the input data if they are present.
Otherwise, the deployment's environment variables are used.
If those are also absent, use defaults.
The API key can only be set using the environment variables.
```

bucket_keys(self)
```
Gets the list of keys in the bucket.

:return: Returns an array of keys.
```

bucket_keys_operation(self)
```
Returns 'invert_boolean' as operation to be used by 'sendTransaction'.
See documentation of 'invert_boolean'.
```

delete(self, key)
```
Deletes a key.
Deleting a non-existent key results in an error.

:param key: The key to delete.

:return: Returns nothing.
```

delete_operation(self, key)
```
Returns 'delete' as operation to be used by 'sendTransaction'.
See documentation of 'delete'.
```

deregister_bucket(self, buck=None)
```
Deletes a bucket.
If buck is defined, deletes a bucket with buck as id.
Otherwise default to current bucketId as initialised in the constructor.

:param buck: Optional bucket id.

:return: Returns nothing.
```

empty_list(self, key)
```
Empty a list.
Removes all elements from the list.
If key is not a list, raises error.

:param key: The key of the list

:return: Returns nothing.
```

empty_list_operation(self, key)
```
Returns 'empty_list' as operation to be used by 'sendTransaction'.
See documentation of 'empty_list'.
```

exists(self, key, otherwise=None)
```
Checks if a key exists.

If otherwise is defined and the key does not exists, set the key to the value of otherwise.
In this case, the value returned will still be false, indicating that it did not exist.

:param key: The key of the data to set.
:param otherwise: The value to set if key does not exist.

:return: Boolean
```

exists_operation(self, key, otherwise=None)
```
    Returns 'exists' as operation to be used by 'sendTransaction'.
    See documentation of 'exists'.
```

export_config(self)
```
    Export the current state configuration.
    Should be used at the end of the deployment to return the state configuration

    :param op: Operation to execute

    :return: Result of the operation
```

get(self, key, delete=None, default=None)
```
    Gets the value associated to the given key.
    The key uses [] to retrieve list and map elements.
    For example:
        arr[1] will retrieve the second element of list arr.
        obj[x] will retrieve the child with key 'x' from the obj map.

    The keys can be nested.
    For example:
        If the second element of arr is a map, arr[1][x] wil retrieve its child 'x'

    This function can be used to initialise values by using the default parameter to set a key if it does not exist yet.
    The other option for initialising values is by using the 'exists' operator with the 'otherwise' param.

    :param key: The key of the data to retrieve.
    :param delete: If set to true, the value is deleted after it is retrieved.
    :param default: If default is set and the key does not exist, the value in default is set in the state and returned.

    :return: The value of the key, if it exists. Otherwise raises error.
```

get_operation(self, key, delete=None, default=None)
```
    Returns 'get' as operation to be used by 'sendTransaction'.
    See documentation of 'get'.
```

increment_number(self, key, value)
```
    Increments a number by a certain amount.
    If key is not a number, raises error.

    :param key: The key of the number
    :param value: The value to increment by, can also be negative

    :return: Returns the new value
```

increment_number_op(self, key, value)
```
    Returns 'increment_number' as operation to be used by 'sendTransaction'.
    See documentation of 'increment_number'.
```

invert_boolean(self, key)
```
    Inverts a boolean.
    Inverting a non-existent key or an invalid datatype results in an error.

    :param key: The key to invert.

    :return: Returns the new value if successfull. Raises an error otherwise.
```

invert_boolean_operation(self, key)
```
    Returns 'invert_boolean' as operation to be used by 'sendTransaction'.
    See documentation of 'invert_boolean'.
```

list_append(self, key, value)
```
    Appends an element to a list.
    If key is not a list, raises error.
    There are no restrictions on the types of the list elements.

    :param key: The key of the list
    :param value: The value to append

    :return: Returns nothing.
```

list_append_operation(self, key, value)
```
    Returns 'list_append' as operation to be used by 'sendTransaction'.
    See documentation of 'list_append'.
```

list_prepend(self, key, value)
```
    Prepends an element to a list.
    If key is not a list, raises error.
    There are no restrictions on the types of the list elements.

    :param key: The key of the list
    :param value: The value to append

    :return: Returns nothing.
```

list_prepend_operation(self, key, value)
```
    Returns 'list_prepend' as operation to be used by 'sendTransaction'.
```

```
See documentation of 'list_prepend'.
```

list_size(self, key)
```
Returns the length of a list.
If key is not a list, raises error.

:param key: The key of the list

:return: Returns the lenght of the list.
```

list_size_operation(self, key)
```
Returns 'list_size' as operation to be used by 'sendTransaction'.
See documentation of 'list_size'.
```

list_trim_left(self, key, value)
```
Trims a list to a certain size.
Removes elements from the start of the list.
If key is not a list or value is not an integer, raises error.
There are no restrictions on the types of the list elements.

:param key: The key of the list
:param value: The length the list should be trimmed to

:return: Returns nothing.
```

list_trim_left_operation(self, key, value)
```
Returns 'list_trim_left' as operation to be used by 'sendTransaction'.
See documentation of 'list_trim_left'.
```

list_trim_right(self, key, value)
```
Trims a list to a certain size.
Removes elements from the end of the list.
If key is not a list or value is not an integer, raises error.
There are no restrictions on the types of the list elements.

:param key: The key of the list
:param value: The length the list should be trimmed to

:return: Returns nothing.
```

list_trim_right_operation(self, key, value)
```
Returns 'list_trim_right' as operation to be used by 'sendTransaction'.
See documentation of 'list_trim_right'.
```

mapKeys(self, key)
```
Returns all of the keys in a map.
If key is not a map, raises error.

:param key: The key of the map

:return: Returns a list of keys
```

map_keys_operation(self, key)
```
Returns 'map_keys' as operation to be used by 'sendTransaction'.
See documentation of 'map_keys'.
```

multiply_number(self, key, value)
```
Multiplies a number by a certain amount.
If key is not a number, raises error.

:param key: The key of the number
:param value: The value to multiply by, can also be negative

:return: Returns the new value
```

multiply_number_operation(self, key, value)
```
Returns 'multiply_number' as operation to be used by 'sendTransaction'.
See documentation of 'multiply_number'.
```

register_bucket(self, buck=None)
```
Creates a bucket.
If buck is defined, creates a bucket with buck as id.
Otherwise default to current bucketId as initialised in the constructor.
If the bucket already exists, does nothing and raises no errors.

Executing an operation on a non-existent bucket will create the bucket automatically.
In the current version this function is not necessary to use state.

:param buck: Optional bucket id.

:return: Returns nothing.
```

send_operation(self, op)
```
Executes an operation.

:param op: Operation to execute

:return: Result of the operation
```

send_transaction(self, ops)

```
Batches multiple operations into a single transaction.
Can be used to reduce the amount of calls to the server.

:param ops: List of operations

:return: Returns the results of the operations in an array
```

set(self, key, value)

```
Sets a key to a value.
The value can be a complex, nested object or list.
If a key already existed, the value will be overwritten regardless of the previous type.

:param key: The key of the data to set.
:param value: The value to set.

:return: Returns nothing.
```

set_operation(self, key, value)

```
Returns 'set' as operation to be used by 'sendTransaction'.
See documentation of 'set'.
```

string_append(self, key, value)

```
Append something to a string.
If key is not a string, raises error.
Value can be string, number or boolean.

:param key: The key of the string
:param value: The string to append

:return: Returns nothing.
```

string_append_operation(self, key, value)

```
Returns 'string_append' as operation to be used by 'sendTransaction'.
See documentation of 'string_append'.
```

string_length(self, key)

```
Get the length of a string.
If key is not a string, raises error.

:param key: The key of the string

:return: Returns the length of the string
```

string_length_operation(self, key)

```
Returns 'string_length' as operation to be used by 'sendTransaction'.
See documentation of 'string_length'.
```

string_prepend(self, key, value)

```
Prepend something to a string.
If key is not a string, raises error.
Value can be string, number or boolean

:param key: The key of the string
:param value: The string to prepend

:return: Returns nothing.
```

string_prepend_operation(self, key, value)

```
Returns 'string_prepend' as operation to be used by 'sendTransaction'.
See documentation of 'string_prepend'.
```

Data descriptors defined here:

__dict__

```
dictionary for instance variables (if defined)
```

__weakref__

```
list of weak references to the object (if defined)
```

Functions

timer = perf_counter(...)

```
perf_counter() -> float

Performance counter for benchmarking.
```

Appendix B

Code examples

```javascript
module.exports = async function(context) {
    // Generate an invocation id and set it in the context
    context.invocationId = uuid.v4();
    // Add monads object to context
    context.monads = {};

    // Dynamically load monads from monads folder based on monads.json
    const monadIds = JSON.parse( fs.readFileSync( join(__dirname, 'monads.json') ) );
    const monads = monadIds.map(m => require(join(__dirname, 'monads', m.id)));

    // Run initialisation of monads
    for (const m of monads) {
      await m.runBefore(context);
    }

    // Execute user function
    const res = await func(context.request.body, context);

    // Run clean-up of monads
    // Adds final monadic configuration to returned monadic body
    const monadicBody = {};
    for (const m of monads) {
      await m.runAfter(context, monadicBody);
    }

    // Return result of the function
    return {
      status: 200,
      body: {
        value: res,
        monads: monadicBody,
      }
    };
}
```

Listing B.1: Implementation of monad layer for Fission in JavaScript

```python
class Deployment:
    def request(self, data, context):
        state = context.state
        results = state.send_transaction([
            state.exists_operation('buffer', otherwise=[]),
            state.list_append_operation('buffer', data['input']),
            state.list_trim_left_operation('buffer', 10),
            state.get_operation('buffer')
        ])

        buff = results[3]

        avg = sum(buff) / len(buff)

        # Return the new average
        return {
            'output': avg
        }
```

Listing B.2: UbiOps deployment that calculates a moving average. Used in evaluation of latency between UbiOps deployments and the shared state prototype.